Cluster scheduling for explicitly-speculative tasks

DAVID PETROU

December 2004

CMU-PDL-04-112

Dept. of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Thesis committee

Garth A. Gibson, chair
Gregory R. Ganger
Srinivasan Seshan
Thomas E. Anderson, Univ. of Washington

This research is sponsored by member companies of the Parallel Data Laboratory Consortium, by a National Science Foundation ITR grant, and by the Army Research Office (contract DAAD19-02-1-0389). The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions in this document are the author's and should not be interpreted as representing the official policies or endorsements, either expressed or implied, of any supporting organization or the U.S. Government.

Report Documentation Page

Form Approved
OMB No. 0704-0188

Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

1. REPORT DATE **DEC 2004**	2. REPORT TYPE	3. DATES COVERED **00-00-2004 to 00-00-2004**

4. TITLE AND SUBTITLE
Cluster scheduling for explicitly-speculative tasks

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Carnegie Mellon University,Dept. of Electrical and Computer Engineering,Parallel Data Laboratory,Pittsburgh,PA,15213

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT
see report

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**286**	

Categories and Subject Descriptors: D.4.1 [Operating Systems]: Process Management — *Scheduling*
General Terms: Algorithms, Design, Performance
Keywords: speculative scheduling, optimistic scheduling, cluster computing, grid computing

Imagine homemade sandwiches.

Abstract

A process scheduler on a shared cluster, grid, or supercomputer that is informed which submitted tasks are possibly unneeded speculative tasks can use this knowledge to better support increasingly prevalent user work habits, lowering user-visible response time, lowering user costs, and increasing resource provider revenue.

Large-scale computing often consists of many speculative tasks (tasks that may be canceled) to test hypotheses, search for insights, and review potentially finished products. For example, speculative tasks are issued by bioinformaticists comparing DNA sequences, computer graphics artists rendering scenes, and computer researchers studying caching. This behavior — exploratory searches and parameter studies, made more common by the cost-effectiveness of cluster computing — on existing schedulers without speculative task support results in a mismatch of goals and suboptimal scheduling. Users wish to reduce their time waiting for needed task output and the amount they will be charged for unneeded speculation, making it unclear to the user how many speculative tasks they should submit.

This thesis introduces 'batchactive' scheduling (combining batch and interactive characteristics) to exploit the inherent speculation in common application scenarios. With a batchactive scheduler, users submit explicitly-labeled batches of speculative tasks exploring ambitious lines of inquiry, and users interactively request task outputs when these outputs are found to be needed. After receiving and considering an output for some time, a user decides whether to request more outputs, cancel tasks, or disclose new speculative tasks. Users are encouraged to disclose more computation because batchactive scheduling intelligently prioritizes among speculative and non-speculative tasks, providing good wait-time-based metrics, and because batchactive scheduling employs an incentive pricing mechanism which charges for only requested task outputs (i.e., unneeded speculative tasks are not charged), providing better cost-based metrics for users. These aspects can lead to higher billed server utilization, encouraging batchactive adoption by resource providers organized as either cost- or profit-centers.

Not all tasks are equal — only tasks whose outputs users eventually desire matter — leading me to introduce the ‘visible response time’ metric which accrues for each task in a batch of potentially speculative tasks when the user needs its output, not when the entire batch was submitted, and the batchactive pricing mechanism of charging for only needed tasks, which encourages users to disclosure future work while remaining resilient to abuse. I argue that the existence of user think times, user away periods, and server idle time makes batchactive scheduling applicable to today’s systems.

I study the behavior of speculative and non-speculative scheduling using a highly-parameterizable discrete-event simulator of user and task behavior based on important application scenarios. I contribute this simulator to the community for further scheduling research.

For example, over a broad range of realistic simulated user behavior and task characteristics, I show that under a batchactive scheduler visible response time is improved by at least a factor of two for 20% of the simulations. A batchactive scheduler which favors users who historically have desired a greater fraction of tasks that they speculatively disclosed provides additional performance and is resilient to a denial-of-service. Another result is that visible response time can be improved while increasing the throughput of tasks whose outputs were desired. Under some situations, user costs decrease while server revenue increases. A related result is that more users can be supported and greater server revenue generated while achieving the same mean visible response time. A comparison against an impractical batchactive scheduler shows that the easily implementable two-tiered batchactive schedulers, out of all batchactive schedulers, provide most of the potential performance gains. Finally, I demonstrate deployment feasibility by describing how to integrate a batchactive scheduler with a popular clustering system.

I have the fury of my own momentum.
Bob, *Fire Walk With Me*

Acknowledgements

I thank Garth Gibson, my thesis advisor, for guiding my intellectual development with wisdom and patience. Garth taught me to ask the right questions and have defensible plans for answering them while giving me freedom to pursue problems interesting to me. Greg Ganger has been a second advisor, providing resources and dispensing advice. Both Garth and Greg have been supportive when crises caused me to take breaks. Tom Anderson was my undergraduate advisor at UC Berkeley and my research advisor in the Berkeley NOW Project. His words of encouragement, many years ago, constantly motivate me. I thank Srini Seshan for being on my thesis committee. I have been lucky to be advised by good people, in mind and heart.

I thank the members of the Parallel Data Laboratory (PDL), especially Garth for creating and Greg for further promoting and developing this institution, with its outstanding intellectual, personal, computational, administrative, and economic resources. The following current and past members of the PDL Consortium provided support: 3Com, Compaq, EMC, Hewlett-Packard, HGST, Hitachi, IBM, Intel, LSI Logic, Microsoft, Network Appliance, Novell, Oracle, Panasas, Quantum, Seagate, StorageTek, Sun, Veritas, & Wind River. Guests at PDL retreats expressed interest in and offered insights for my research. PDL staff members Joan Digney, Jennifer Landefeld, Karen Lindenfelser, & Patty Mackiewicz provided a positive work environment. Other PDL and ECE staff members supported my computing resources.

Sharing 8208 Wean with Jason Flinn, Dushyanth Narayanan, & Sanjay Rao was often educational and always fun, despite music selection disagreements. Early on, Khalil Amiri was friend, elder gradsperson, and research collaborator. I profited from communicating with Sonya Allin, Mor Harchol-Balter, Miron Livny, Andy Myers, Jiri Schindler, & Steve Schlosser. Exchanging ideas was a bonus to my friendships with Sourav Ghosh, John Griffin, Dushyanth Narayanan, David Rochberg, Craig Soules, Eno Thereska, David Tolliver, & Jay Wylie. My time as an EECS undergraduate at UC Berkeley was pleasantly passed with Will Chow, Daniel (‘danh’) Holliman, John Milford, Sameer Parekh, & Ali Rahimi. Remzi Arpaci-Dusseau, Doug Ghormley, Brian Harvey, Carlo Séquin, & Amin Vahdat were inspirations.

My Pittsburgh years have been happy, a time of varied experience and personal growth. I owe this being close to Dan Baselj, Julie Brick, Ben Feldman, Mark Lazarev, April Murphy, Dushyanth Narayanan, Hille Marika Paakkunainen, Jill Penman, Megan Schmidgal, David Tolliver, & Jay Wylie.

Thanks for the existence of the 61C Cafe, where I was found holding court, courting, coding, writing, composing, and enjoying company. Baristas of note include Jason Bacasa, Keith Kaboly, Moshe Marvit, Nick Sarno, & Danielle Skoncey. Crazy Mocha's Leah Loyd, Deanna Mance, & Dana Waelde generously hosted me during the final months.

From California, my first best friends and bandmates I acknowledge: Ean Brown, Brian Gilmore, & Huy Huynh. Highschool friends shape each other, and I was glad to know Derald Brenneman, Joy(zelle) Davis, Sheila Salamipour, Kevin Stephenson, (the late) Stuart Tay, & Jason Thibodeau. My current Pittsburgh bandmates Hille Marika Paakkunainen and Mike Shanley provide an opportunity to play again. From the Music Department at Carnegie Mellon University, Nancy Galbraith, Natalie Ozeas, Marilyn Taft-Thomas, Donald Wilkins, & Colette Wilkins and the Dalcroze Eurhythmics faculty taught me and encouraged my musical aspirations.

Closest to me are my late mom, my dad, sister, brother(-in-law), & niece, all from whom I receive overwhelming and unconditional love. Nothing in my life would work without them. My extended Italian and Greek families are also a source of love and support. I love you all.

And thanks to the Allegheny Cycling Association for providing a conduit for reoccuring, volcanic bursts of energy, and to Danny Chew for being the greatest bicycling inspiration.

David Petrou · Pittsburgh, Pennsylvania · December, 2004

Contents

Figures

Tables

> There is nothing more difficult to take in hand,
> more perilous to conduct, or more uncertain in
> its success, than to take the lead in the
> introduction of a new order of things.
>
> Niccolo Machiavelli, *The Prince*

1 Introduction

A process scheduler on a shared cluster, grid, or supercomputer that is informed which submitted tasks are possibly unneeded speculative tasks can use this knowledge to better support increasingly prevalent user work habits, lowering visible response time (the time between needing and receiving task output irrespective of when a task was submitted), lowering user costs, and increasing resource provider revenue.

Large-scale computing often consists of many speculative tasks to test hypotheses, search for insights, review potentially finished products. Tasks often sit in queues for a long, unpredictable amount of time. This thesis addresses how to reduce or eliminate visible response time by prioritizing work that a user is or will likely soon be waiting on and wasting fewer resources on speculative tasks quite likely to be canceled.

Imagine a scientist using a shared computing cluster to validate a hypothesis (Figure 1.1). She submits chains of tasks that could keep the system busy for hours or longer. Tasks listed earlier are to answer pressing questions while those later are more speculative. Early outputs could cause the scientist to reformulate her line of inquiry; she would then reprioritize tasks, cancel later tasks, issue new tasks. Moreover, the scientist is not always waiting for tasks to complete; she spends minutes to hours studying the output of completed tasks, attends meetings and lunches, and stops working as evening approaches.

On existing schedulers without speculative task support, this behavior results in a mismatch of goals and suboptimal scheduling. Should a user who does not know which tasks will bear fruit submit one speculative task, a few, many, or every conceivably useful task? After all, defining speculative tasks is a time-consuming activity in itself. A user wishes to reduce the time waiting for needed task output, increasing the rate at which scientific inquiry is accomplished, and reduce the amount charged for unneeded speculation. The right amount of speculation depends on considerations difficult and burdensome or impossible for a user to know, including to what extent

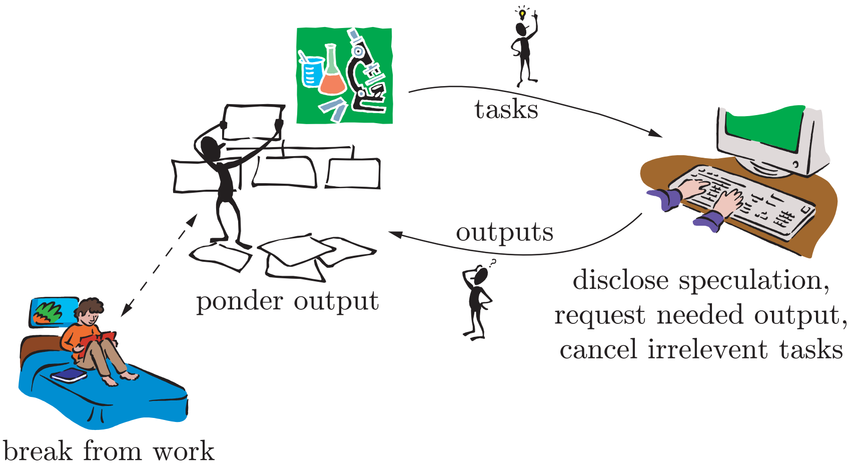


Figure 1.1: Speculative user behavior. While performing computationally intensive research, users wish to pipeline the execution of chains of speculative — not known to be needed — tasks with the consideration of received task outputs and optional periods of rest. This thesis removes the barriers presented by existing cluster scheduling to exploiting this way of working.

a task is in fact speculative and the behavior of other users. In situations in which resources are not directly charged or users have the means to pay for wasted work, users might overwhelm resources with speculative tasks in an attempt to reduce delay. This thesis addresses these and other concerns with 'batchactive' scheduling solutions (combining batch and interactive characteristics) to exploit the inherent and easily disclosed speculation in common application scenarios.

With a batchactive scheduler, users submit explicitly-labeled chains or batches of speculative tasks exploring ambitious lines of inquiry, and users interactively request task outputs when they are found to be needed. After receiving output and considering this output for some time, a user decides whether to request more outputs, cancel tasks, or disclose new speculative tasks. Users are encouraged to disclose more computation because batchactive scheduling intelligently prioritizes among speculative and non-speculative tasks, providing good wait-time-based metrics, and because batchactive scheduling employs an incentive pricing mechanism that charges for only requested task outputs (i.e., unneeded speculative tasks are not charged), providing better cost-based metrics for users. These aspects can lead to higher billed server utilization, encouraging batchactive adoption by resource providers organized as either cost- or profit-centers.

Speculation to improve performance is a pervasive concept in computer systems found at the level of I/O requests, program blocks, instructions across all areas of computing including architecture, languages, systems. In this introduction chapter I begin by examining the ways in which certain work habits, e.g., conducting semi-interactive exploratory searches, can provide or already provide speculative tasks, and I discuss the mismatch between this work and traditional processor scheduling. Following this, I sketch my approach to scheduling. I then state my thesis and foreshadow my contributions. Before ending this chapter, I state the organization of the rest of this dissertation.

· · ·

Users can often plan ahead, pipelining the consideration of received task outputs with the execution of speculative tasks whose outputs were not known to be needed at the time of submission. Important applications consist of speculative tasks and intelligently scheduling these tasks is increasingly important in clusters, grids, and supercomputers.

Scientific disciplines and commercial ventures use shared computer resources to simulate phenomena, evaluate hypotheses, visualize information, discover invariants. Researchers in high-energy physics, cosmology, seismology, weather forecasting, aerodynamics use computing resources speculatively. Scientists in national laboratories, academic institutions, private research departments often construct series of experiments in the advancement of science occupying considerable computing time, in which at the outset it is unclear which task outputs in such exploratory searches will be useful.

The following are examples of users submitting sets of speculative tasks when performing exploratory searches or parameter studies. My scheduling solutions apply to such processor-based, non-parallel examples.

- Bioinformatics comprises the methods for solving nucleotide sequencing problems such as constructing a genome out of fragments and determining protein function. Part of solving such problems is comparing new sequences to known sequences. Bioinformaticists share workstation farms for performing sequencing tasks. A batchactive scheduler would enable scientists to explore ambitious biological hypotheses without fear that resources would be wasted on speculative chains of work that might be canceled after early outputs were scrutinized.

- Computer animation is increasingly used in motion pictures. Teams of artists creating a computer-animated film submit scenes for rendering to clusters. This work is highly speculative. Upon seeing initial frames

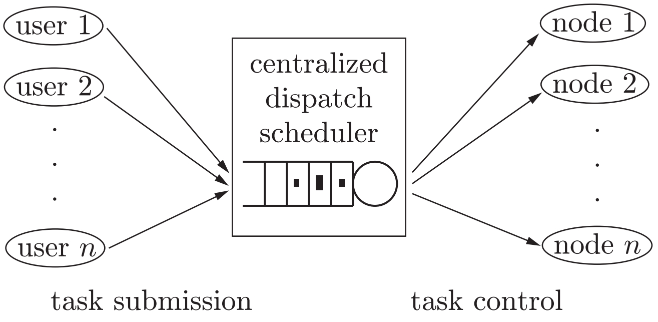


Figure 1.2: Centralized cluster scheduler. Users send task requests and cancelations to a centralized cluster scheduler. This scheduler orders and distributes work among multiple cluster nodes. Task outputs are written to a shared store (not shown) accessible by the submitting user.

(computed by a chain of tasks), an artist may decide that a rendered object could be in a better location, e.g. With a batchactive scheduler, artists could prioritize key sections of a scene, those with more action, e.g., to more quickly decide whether additional frames are worth having. If it becomes known that unviewed, possibly uncomputed, frames will not be needed, artists would cancel the renderings of these frames to free resources for other rendering tasks.

- Computer scientists routinely share clusters to run simulations exploring high dimensional spaces. Parameter studies for feature extraction, search, or function optimization can continue indefinitely, homing in on areas for accuracy or randomly sampling points for coverage. Simulations are used to study, among other things, microarchitecture cache behavior, computer virus propagation, and I/O storage patterns. With a batchactive scheduler, such chains of simulations could occur in parallel with experimenters analyzing desired and completed outputs and guiding the searches in new directions, canceling branches determined to not be useful. Speculative simulations would operate in the background when pressing outputs are needed.

Clustering software provisions the resources of multiple nodes to multiple users. Users send task requests and task cancelations to a centralized scheduler employing a policy to distribute work among nodes toward meeting some combination of time and cost goals for users and the resource provider. This organization is depicted in Figure 1.2.

Existing cluster, grid, supercomputer scheduling, which in practice is a

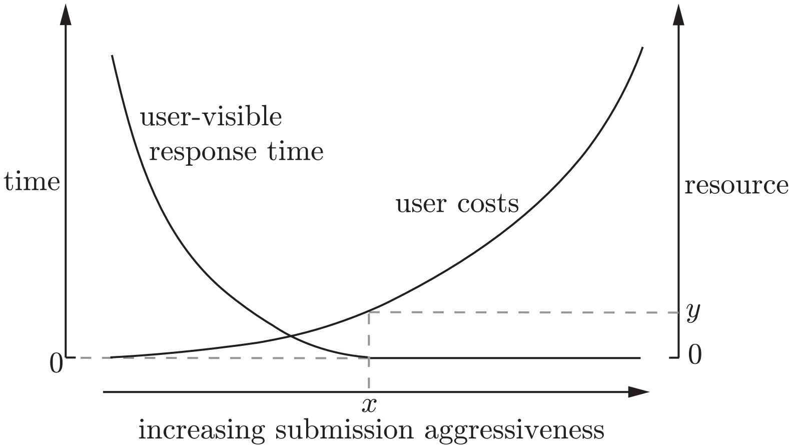


Figure 1.3: A sketch of how submission aggressiveness affects visible response time and user costs under an existing pricing mechanism in which all resource usage is charged. The more speculative tasks a user submits, the less visible response time he or she will experience for any task later deemed to be needed, but the more he or she will pay for larger numbers of unneeded speculation. With sufficient think time and deep speculative queues, as shown for submission aggressiveness greater than x, it is possible to eliminate visible response time. The lowest cost to the user that achieves this is denoted by y. However, unknown or difficult to predict run-time considerations prevent a user from making such time / cost tradeoffs. Note that the two vertical axes are of different units.

variant of decay-usage or first-come-first-serve, does not know which tasks are speculative and thus cannot schedule them differently from tasks that are known to be needed. Computing time is either sold to another party (under names such as 'third-party compute outsourcing' and 'information technology resource providers') or the resource owner and user are the same person, organization, or entity and computing time is not directly charged. When sold, all resource consumption, whether or not speculative task outputs are eventually determined to be needed, is charged.

Should a user exploring a space speculatively submit one task, a few, many, or the entire 'computational plan?' When resources cost, the user is pressured to only submit a few tasks at once because the user does not wish to be charged for running tasks whose outputs might be determined to be unneeded. But doing so leads to poor time-based scheduling metrics, such as visible response time, because the speculative tasks are not executing concurrently with the user's think time of received task outputs as much as possible. This tradeoff is illustrated in Figure 1.3. When resources are not directly charged (such as in a communal cost-center), or if the user is willing to pay for unneeded tasks, then the user should submit many speculative tasks so that they might execute before being needed. However, if every user did this, then resources would be overwhelmed with speculative tasks and the response time for non-speculative tasks executed after many eventually useless tasks will increase dramatically.

In small communities, users would like to appear to submit a 'reasonable' number of speculative tasks, in hopes of balancing their wasted costs and response time with the needs of other users. However, even if everyone wished to cooperate, there is no clear way for a user to determine which and how many speculative tasks to submit. Meeting individual and collective goals depends on many unknown factors: the pattern of other task arrivals, task service time, user think time, and the probabilities that speculative tasks will be needed.

These problems cannot be overcome without a scheduler that discriminates between speculative and non-speculative tasks. Batchactive scheduling assumes this ability.

Batchactive scheduling leverages existing opportunities to better schedule speculative tasks. The existence of think time gives a batchactive scheduler the flexibility to defer the execution of non-pressing, speculative tasks in favor of known needed or likely to be needed tasks. Since speculative tasks might be canceled, delaying their execution might result in eventually unneeded tasks being canceled before they consume significant, if any, resources; deferred work can be saved work. Related to think time is a concept that can be similarly leveraged that I call 'away periods,' reflecting when people become unavailable to consume task output independent of task completion — e.g., a user leaving at the end of the work day and not being ready to consider output until the next morning. Moreover, spare computational resources, which are available in many settings, can be exploited for executing speculative tasks. Once users have the means provided by batchactive scheduling to convey to the system which tasks are speculative v. needed, even more resources will be available to obtain the benefits of batchactive scheduling.

· · ·

Batchactive scheduling is intended for shared clusters, the most important architecture for high-end computing (Figure 1.4). Clusters are cost-effective and flexible, used for small computing resources, computational grids, and supercomputers. The positive results of this thesis can have immediate impact by being deployed as extensions to clustering software such as Condor, Platform LSF, the Globus Toolkit, Legion (Avaki), and the Sun ONE Grid Engine.

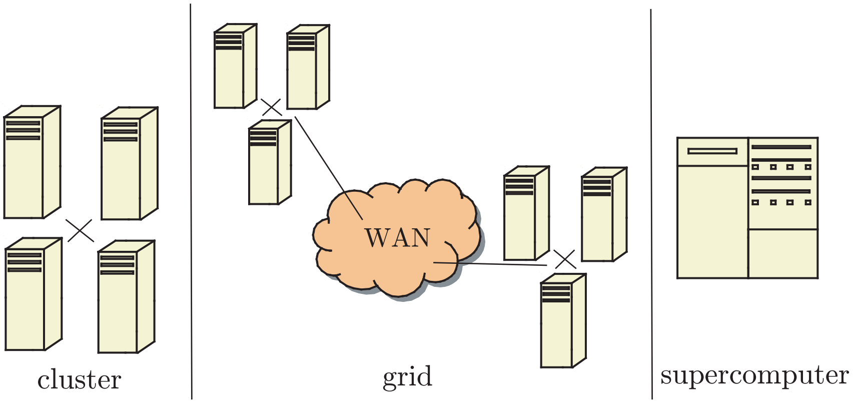


Figure 1.4: The target cluster architecture. Clusters, a loose to tight collection of nodes, are a cost-effective and flexible solution for small- to large-scale computing. A grid is a collection of possibly geographically separate clusters accessed through a wide area network. Traditional supercomputers are tighter aggregations of a large number of processor nodes. Clustering software, which can be extended with batchactive policies, manages user workloads in the form of tasks.

In batchactive scheduling as I define it, users judge tasks as either speculative or needed, and speculative tasks are organized in some structure, such as a list (chain), directed acyclic graph, or with no ordering constraints. There are no 'levels' or probabilities of speculation, which would be a burden for users to predict and provide. Users disclose speculative tasks and request tasks whose outputs they know they need. A user may cancel any task if received outputs suggest their irrelevance. I call this 'batchactive' usage of the system, because, like batch usage, many tasks are submitted at once, and, like interactive usage, the user is waiting for the output of (usually) one identified task. (However, unlike batch usage, the scheduler knows which tasks are speculative, and unlike interactive usage, entire sets of speculative tasks belonging to one user are often in the system at once.)

Speculative disclosure is a form of hinting which only reveals user expectation, enabling the system to globally optimize resource management. These hints express information independent of system implementation, remaining correct if the environment changes. The disclosure interface, being the same as for requesting non-speculative tasks, should also be easy to use.

Endowing the scheduler with the knowledge of tasks that may be needed in the future enables servers to get an early start, rather than being idle, while preventing speculative tasks from overwhelming the system. Further, knowing about speculative tasks exposes parallelism from a user's workload when the execution of these tasks do not depend on outputs from one another. Such speculative tasks can leverage the parallelism of cluster nodes.

The batchactive incentive pricing mechanism diverges from the norm of charging for all resource usage. Disclosed tasks that were never needed are not charged. With this mechanism, the user does not need to weigh the estimated cost (wasted money) and benefit (better visible response time) of each disclosure, encouraging the user to freely disclose tasks. Servers that are either cost-centers (non-profit) or profit-centers, covering most organizations, can be motivated to institute the batchactive pricing mechanism: in a cost-center, the cost for requested (needed) tasks can be adjusted so that total billing over some time is the same as in the traditional pricing mechanism, and, in a profit-center, improved time-based metrics coupled with no risk for the user to disclose speculation can encourage more users, the submission of longer chains of tasks, and larger tasks, resulting in higher billed server utilization.

The traditional response time metric conflates the time a task was submitted with the time a task's output was needed. That is, traditionally systems measure time from submission to completion of a task regardless of when its user needs its output. I introduce 'visible response time,' the time between needing (wanting to begin to use) and receiving task output, or the time 'blocked on' output. Visible response time accrues only after a user asks for output from a task that may have been submitted much earlier and thus measures the time that a user actually waits for output, which is usually less than the time that a speculative task has been in the system. In particular, a task can and often does have zero visible response time if it was speculatively disclosed and was completed while its user was examining the output of some other task. Other metrics, such as visible slowdown, are derived from visible response time.

Most batchactive schedulers in this dissertation share the property that requested tasks have absolute priority over speculative tasks. (More complex but harder to deploy policies are also discussed.) This prioritized two-tiered approach — having independent queues for requested and disclosed work and shown in Figure 1.5 in contrast to policies which do not discriminate — is sufficient most of the time. Two out of five disclosed queue subpolicies are novel. One favors users who speculate less (i.e., users who submit speculative tasks that are more often found to be needed), while the other favors users who have requested (paid for) more work.

Speculative scheduling may be compared to the concept of delayed or lazy evaluation found in programming languages. In delayed evaluation, only desired outputs from an unbounded computation (e.g., an infinite list) are

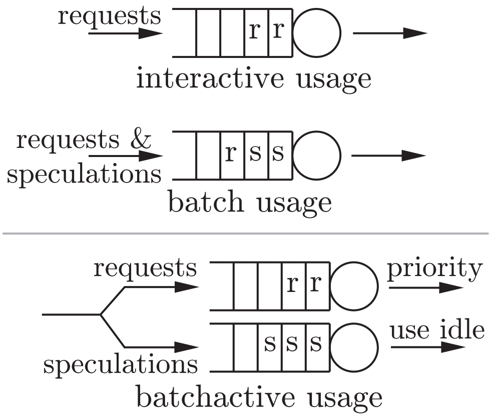


Figure 1.5: Comparison of the usage of non-speculative and batchactive scheduling. The top two queues illustrate two extreme behaviors of users using a non-speculative scheduler. Needed tasks are requested one at a time (interactive) and needed and speculative tasks are requested in sets (batch). The bottom queue is a two-tiered batchactive scheduler which gives priority to non-speculative tasks. Users behaving in a batchactive manner submit both needed and speculative tasks. In contrast to batch usage, users label which tasks are speculative so that the scheduler can treat them differently, to prevent speculative tasks from starving non-speculative tasks.

actually computed. Batchactive scheduling also defers work but often computes outputs not known to be needed. If speculative tasks are executed only after a user desires their outputs, as in delayed evaluation, then there is no performance benefit to speculative task disclosure. However, at the other extreme, if speculative tasks execute with the same priority as non-speculative tasks, then system resources will be squandered with possibly unneeded work. How to balance between executing speculative tasks within a user's and among users' tasks (i.e., avoiding self-interference and cross-interference) is one goal of this thesis; how to execute a speculative task at the last moment when this moment is unknown.

· · ·

My thesis is that a multiuser process scheduler informed of which submitted tasks are speculative can provide better time- and cost-based metrics for users and resource providers. I provide evidence for the following elaborations throughout this dissertation:

- there exists a class of applications in which work is submitted speculatively and that this class is important and will become more so (Chapter 2.2);

- speculative tasks are poorly exploited by existing schedulers (Chapters 2.4, 4.8, and 6.2.2);
- speculative task disclosure and the batchactive pricing mechanism support how people wish to work for many application scenarios, including their desire to pipeline think time and task execution (Chapters 2.1, 5.3, and 5.1);
- in a single-server simulation, batchactive scheduling can substantially reduce visible response time (among other time-based metrics), reduce user costs, and in some cases improve resource provider revenue (Chapter 6.2);
- two-tiered batchactive scheduling is effective, deployable, and exhibits low overhead (Chapters 7.2 and 7.1.4).

For the scheduling researcher, speculation requires rethinking metrics and algorithms: not all tasks are equal — only tasks whose outputs users eventually desire matter — leading me to introduce the 'visible response time' metric and the batchactive pricing mechanism which is resilient to abuse. I argue that the existence of user think times, user away periods, and server idle time makes batchactive scheduling applicable to today's clusters, grids, and supercomputers.

I study the behavior of speculative and traditional scheduling through the simulation of a model of users, tasks, and a single server based on important application scenarios. I created this highly-parameterizable discrete-event simulator and contribute it to the community for further scheduling research.

I answer several specific questions: How do the simplest, most easily deployable batchactive schedulers compare to the simplest commonly used non-speculative schedulers? Can novel scheduling subpolicies for speculative tasks leverage historical user patterns to achieve to better performance? How do such subpolicies compare to each other and to non-speculative scheduling? How does the improvement of batchactive scheduling over non-speculative scheduling change when utilizing task size information, which may be available through prediction? To what extent can an oracular scheduler perform even better? Can batchactive scheduling simultaneously lower user costs on unneeded speculation and increase server revenue by improving server utilization? At the same user costs, can visible response time be reduced? At the same visible response time, can user costs be reduced?

For example, over a broad range of simulated user behavior and task characteristics, I show that under a batchactive scheduler visible response time is improved by at least a factor of two for 20% of the simulations using schedulers based on first-come-first-serve. (For some deployed traditional batch schedulers based on resource usage, the performance difference is not as pronounced, but batchactive scheduling still wins.) On a non-speculative scheduler, there are extreme situations (such as high load) in which users who submit one task at a time results in better performance than users who submit batches of tasks at a time. While at other extremes (such as low load), the opposite is true. But users submitting work to a batchactive scheduler results in as good or better performance than non-speculative scheduling for both these extremes and better performance for intermediate situations, exhibiting adaptability. Another result is that visible response time can be improved without decreasing the throughput of tasks whose outputs were desired. Under some situations, user costs decrease while server revenue increases. Related is that more users can be supported and greater server revenue generated at the same mean visible response time. Further, two-tiered batchactive schedulers that are simple, out of all batchactive schedulers, provide the bulk of the potential performance gains.

I examine the circumstances regarding task characteristics and user behavior in which batchactive scheduling provides the best results versus when it performs similarly to non-speculative scheduling. Some experiments illuminate the non-obvious necessity of user think time to provide speculative scheduling benefits. Another finding is that my approach applies best when several to a potentially unbounded number of speculative tasks are submitted. Considerable performance improvements are found even when the average length of a user's speculative tasks is three or four.

I demonstrate deployment feasibility by describing how to integrate a batchactive scheduler with a popular clustering system called Condor. I also measure scheduling overhead and show it to be negligible.

I establish these and other aspects and arguments for batchactive scheduling in the following order. Chapter 2 motivates batchactive scheduling by describing prevalent work patters, important applications, and opportunities for better scheduling. It also discusses related work in the use of scheduling speculation across tasks and within tasks. Chapter 3 states what is inside and outside the scope of my thesis. My scope encompasses important application scenarios while avoiding issues orthogonal to how the knowledge of whether a task is speculative can be used to improve scheduling metrics. In Chapter 4 I provide non-speculative cluster scheduling background, including a description of the target architecture, standard cost models, user and resource provider goals, and fundamental and commonly-deployed scheduling policies.

Chapter 5 introduces batchactive scheduling, the new batchactive pricing mechanism, new scheduling metrics based on visible response time, ambitious policies requiring difficult to obtain information, and two-tiered policies that are deployable. This chapter also discusses speculative scheduling beyond centrally scheduled processor resources. The simulation results are in Chapter 6, which details the simulation model, the ranges of simulated behaviors, and the differences in performance between non-speculative and batchactive schedulers. Chapter 7 details the design and implementation of the simulator and describes how to deploy batchactive scheduling to an existing cluster by extending a popular clustering system.

Finally, Chapter 8 recapitulates the motivation and contributions of this thesis and discusses non-technical challenges to deploying batchactive scheduling.

· · ·

Batchactive schedulers, which recognize speculative tasks as first-class entities, attempt to maximize human productivity and minimize user resource costs by scheduling and charging speculative tasks more effectively. Ambitious user hypotheses potentially requiring an unbounded amount of resources can be explored without fear that resources would be wasted on long-shot speculation. What is required is for users to disclose their speculative plans and then request individual task outputs when it becomes known that these outputs are needed. Existing policies and pricing mechanisms were designed for non-speculative tasks; tasks whose outputs were all known to be needed. However, this is not always true and suboptimal scheduling results when users engaged in speculative searches use non-speculative schedulers.

The cost of cycles decreases while the cost of human time increases, making task speculation more common and speculative scheduling more applicable. The heart of this work is deciding when tasks should run to reduce or eliminate visible response time across users while not wasting contended resources on speculative tasks that might be canceled. Novel policies for speculative tasks reward good science; the better someone is able to specify needed work, the better the scheduler performs for that person.

Opportunity is missed by most people because it
comes dressed in overalls and looks like work.
Thomas Edison

2 Opportunities for batchactive scheduling

This chapter motivates batchactive scheduling (Chapter 5). I begin by describing a prevalent way in which people work. Users can often plan ahead, submitting a number of potentially needed speculative tasks and pipelining the consideration of completed task outputs with the execution of tasks whose outputs are not yet known to be needed. This behavior, in contrast to not disclosing speculation, is depicted in Figure 2.1.

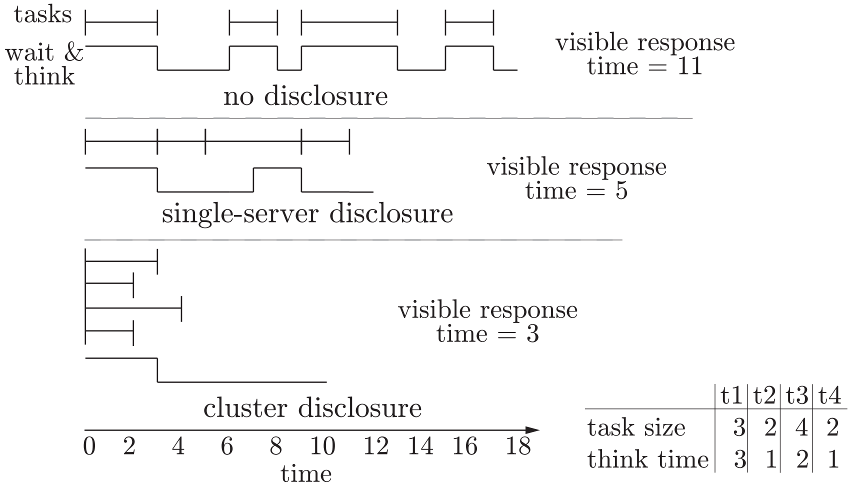
The following sections elaborate on these work patterns showing that there are opportunities for smarter scheduling to aid speculative work. I present actual scenarios of users engaged in these work patterns for diverse applications categorized as exploratory searches, sequential tasks, and parameter studies.

I cite evidence that speculative work is often not known to be actually desired until some time (sometimes a long time) after the work is submitted. I also describe the prevalence of idle computational resources that can be leveraged for speculation.

I then describe how non-speculative schedulers (Chapter 4) handle speculative tasks poorly with respect to time- and cost-based metrics (Chapter 5.2). In contrast to batchactive schedulers, they do not embrace the aforementioned opportunities to provide better performance for speculative tasks. Before summarizing this chapter, I describe related work in applying speculation to the scheduling of tasks, scheduling speculative parts of a single task, and scheduling speculative activity that uses resources other than the processor.

2.1 Work patterns

O'Day and Jeffries [1993] studied how 'information seekers' perform searches. For a number of activities, they found that people tend to conduct a series of interconnected but diverse searches. They studied the behavior of fifteen individuals (including a financial analyst, venture capitalist, marketing en-



	t1	t2	t3	t4
task size	3	2	4	2
think time	3	1	2	1

Figure 2.1: A sketch of how visible response time changes when a user discloses work. Shown is the total visible response times for a single user submitting four tasks and waiting for and thinking about the outputs of these tasks. Visible response time is the time between needing and receiving task output irrespective of when the task was submitted. Three settings are shown. First, the user does not disclose speculative work; he or she behaves interactively, submitting the next task after thinking about the output of the previous task. Second, the user discloses all speculation to a single server. And third, all speculation is disclosed to a cluster with many nodes, allowing all four tasks to run in parallel. Each setting shows the execution of the tasks and the user cycling between waiting and thinking (this cycle is depicted abstractly in Figure 1.1), over time. In each setting, the user thinks and the servers are busy the same total amount of time. What differs is the total visible response time, which improves as the opportunities for pipelining task execution and think time increases. Due to the limitations of existing schedulers, users who disclose will either do so and be charged for unneeded work and wait longer by competing with other users' speculative work, or behave interactively to avoid these risks. Batchactive scheduling encourages the behavior depicted in the bottom two settings with the batchactive pricing mechanism and intelligent scheduling.

gineer, demographer, consultant, statistician, among others), half of which worked for one large computer manufacturer while the other half worked for a variety of other companies. These individuals conducted searches on financial and business-related topics. O'Day and Jeffries' conclusions result from interviews conducted in the offices of these individuals.

The fifteen individuals participated in surveys for each of 66 activities. After the surveys were conducted, one-third of their activities were categorized as exploratory searches. Outputs from such searches are analyzed for trends or correlations, compared against different pieces of data sets, scaled or aggregated, and finally interpreted. One individual reports, 'What you want is a thorough and efficient way that will cover first of all the leading sources and then second of all more localized sources.... So there is a kind of a general quick and dirty way to find out what's out there and then once you do that then you want to go in more specifically and that is where you get into more detailed searches. There is like a $30,000$-foot view and then you go into specific areas.'

These individuals were behaving speculatively and exhibited think time: analysis of preliminary searches may remove the need to consume outputs from detailed searches that may be computational processes operating in the background. Speculative behavior is common on computing systems as shown next (Chapter 2.2) and takes the form of users submitting speculative tasks. A speculative task is some unit of work, usually corresponding to a run of an application, whose output is not yet known to be required [DeGroot, 1990]. Other applications never terminate; they cycle between performing work, delivering output, and idling for the next user directive. Here, the task is the work cycle of the application. The granularity of a task is such that a person uses its output after receiving it, e.g., to make task submission or cancelation decisions. If application output must be processed, filtered, graphed and if such steps can occur without human intervention, it is simpler to consider the aggregate of these operations as a task.

The trends of rising cost of an average person's time and decreasing cost of computational resources make speculative workloads more prevalent. Economists argue that the rise of inflation-adjusted average wage, which has been measured over decades, implies that the time of the average person has become more valuable [Becker, 1965; Romer, 2000; Pashigian et al., 2003; Walker, 2002]. Chip miniaturization and economies of scale are two contributing factors to the cheaper and more available computational power witnessed over decades [Moore, 1965; Gibbs, 1997; Hennessy et al., 2002]. These trends are predicted to continue. Either continuation is sufficient to suggest that users will increasingly risk wasting computational resources to

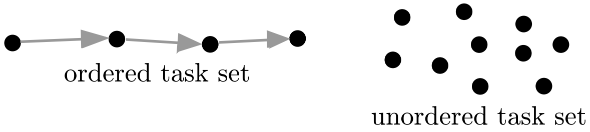


Figure 2.2: Two simple organizations for speculative tasks include flat list order and no ordering preference.

save human time. Together, users are compelled more strongly to increasingly speculate.

I call all the speculative tasks associated with a user his or her *task set.* Task sets may include tasks from several applications or from one application run with different parameters. The user may end up wanting one, all, or some of these tasks. After a task whose output is known to be needed completes, the user receives this output and considers this output for some duration or *think time.* At that point, the user determines whether more output from the task set is needed. If not, it could mean that the user has sufficient, conclusive output. Or it could mean that the task set is not answering the right questions, in which case tasks (that may or may not have completed) whose outputs are undesired are canceled, and a new task set exploring a different line of inquiry is issued. Two simple organizations for tasks in a task set are flat list order or no ordering preference as depicted in Figure 2.2. Unordered desire, similar to the 'dynamic sets' of Steere [1997] (Chapter 2.5.3), reflects users who do not know if any output is more useful than another; any answer is helpful until more is known. Only applications that require some amount of user think time (Chapter 2.3.1) to determine whether more outputs are useful are good candidates for batchactive scheduling,

A common example of speculative work is an application run repeatedly with different arguments to search a large parameter space first in broad stokes, randomly or at specific parameter intervals (iterative or successive refinement or improvement), then in detail at areas of interest (Figure 2.3).

Any-time algorithms (related to imprecise computing) can generate output after using some amount of resources or after achieving some level of quality [Musliner et al., 1992]. The creation of each intermediate output constitutes the work of one task. Figure 2.4 illustrates a hypothetical simulation in this spirit whose output changes across runs, until successive runs do not provide additional information. An actual example is the rendering application in Chapter 2.2.2. An example from the database community presents partial outputs and lets the user guide the search when querying diverse (both in the nature of the content and size), distributed, Internet-based

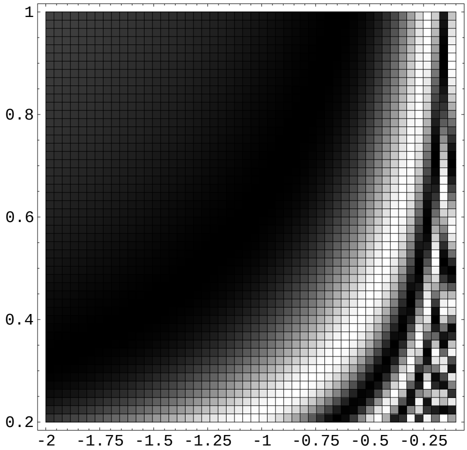


Figure 2.3: A sketch of an initial exploration of a two-dimensional parameter space, indicating regions for further study. After scanning in low detail a parameter space, the user will refine his or her search to the interesting region in the lower-right, canceling speculative work that would produce more detailed results in other regions.

databases [Raman and Hellerstein, 2002].

There can be a large to unlimited number of tasks in a task set, potentially requiring an unbounded amount of computing resources. Examples are scientific 'grand challenges' which are fundamental problems in science and engineering with broad economic and scientific import [Argonne, 2004; UK Computing Research Committee, 2004]. The existence of resource intensive task sets suggests that the longterm relevance of the scheduling policies for speculative workloads I present in this thesis will not be diminished by the speedups in computer hardware predicted by Moore's law [Moore, 1965].

Defining speculative tasks may be a time-consuming activity for a person to perform. Sometimes an autonomous program agent can work on behalf of a user in constructing task sets. An agent can choose tasks in an attempt to anticipate a person's needs. Tennenhouse argued that effective computer use, if architectural and task demand trends continue, will necessitate less interaction between people and computers [Tennenhouse, 2000]. Across computing history, the number of processors per person has gone up. When moving beyond one computer per person, computing paradigms must shift from human-centric to human-supervised. People are serialization points that dampen feedback that could sometimes occur automatically. Tennenhouse's work in 'proactive computing' seeks to remove people from the control loop. Proactive systems will anticipate user needs: excess

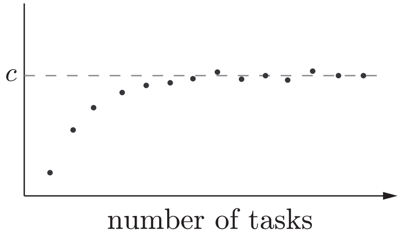


Figure 2.4: A sketch of how successive runs of an any-time application can determine whether more outputs are fruitful. In this example, the more runs, the less useful the outputs, as the outputs converge to some value c. Queued tasks for any-time applications are often speculative: the trend of the output values, which may require user think time to identify, may cause the user to determine that future outputs are unneeded.

computation and communication capacity will be harnessed to fetch and manipulate information, producing answers before they are needed. Examples where an agent designed for a particular domain can automatically construct relevant task sets are in Chapter 2.5.

Although existing schedulers are not designed for them, users often submit speculative tasks; i.e., users disclose their computational plans by submitting tasks whose outputs they do not know whether they need at the time of submission. For example, users engineer tasks to run overnight [Wenisch, 2003]. They do this to increase the chance that they will have useful task output and decrease the chance that they must wait for tasks to complete.

The next several sections expand on the above work patterns to motivate my batchactive scheduling solutions for speculative tasks. I detail scenarios in which people submit speculative computational work, show the existence of think times and a related concept I call 'away periods,' and show that there exists computational capacity in clusters, grids, and supercomputers for running speculative work.

2.2 Scenarios

Scientific disciplines and commercial ventures use computer resources to simulate phenomena, evaluate hypotheses, visualize information, discover invariants. Researchers in high-energy physics, cosmology, seismology, weather forecasting, aerodynamics use computing resources speculatively. Scientists in national laboratories, academic institutions, private research departments often construct series of experiments occupying considerable computing time, in which, at the outset, it is often unclear which task outputs will be useful.

This section presents real-world processor-bound examples showing that computing resources are used for speculative work, motivating the speculative scheduling policies (Chapter 5) central to this thesis. These scenarios are important, both because of the economic scale of the industries in which they are found and because of their place in the advancement of science. They are categorized as exploratory searches, sequential tasks, and parameter studies and concern bioinformatics, computer animation, and computer simulation, respectively. Although the focus of this thesis is on the processor resource (Chapter 3.3), I also describe non-processor-based scenarios for completeness. I obtained much of the following corroborations of the above work patterns through ad hoc surveys and cited personal communications.

2.2.1 Exploratory searches

An *exploratory search* (speculative search, speculative test [DeGroot, 1990]) is typically a hand-crafted chain of speculative tasks from different applications (e.g., process dataset A, filter table B, combine them into $C, \ldots$) whose outputs increasingly provide evidence to confirm or refute a hypothesis. Exploratory searches in the area of information retrieval, called 'berrypicking techniques,' has received attention [Bates, 1990]. The applications suitable for batchactive scheduling are those in which one can form the next search speculatively, before the prior search completes. The scenario for this type of speculative work that I examine concerns bioinformatics.

Bioinformatics comprises the computing methods for solving problems concerning nucleotide sequences. One problem is constructing a complete genome out of fragments. (The gist of the problem is analogous to putting a puzzle together when a flashlight can only shine on several pieces at a time.) A large effort recently sequenced a complete human genome [Genome, 2001]. Another problem is determining protein function, which has implications in measuring susceptibility to and the prevention of disease. (A protein's shape determines its function and nucleotide sequences determine a protein's shape [King, 1993; Thomasson, 2004].) An important part of solving both problems is comparing new sequences to known sequences to find similar structure, function, and origin.

Several algorithms related to substring matching have been adapted to comparing nucleotide sequences for similarities [Karp and Rabin, 1987; Smith and Waterman, 1981]. Sequence similarity is based on biological criteria. Some algorithms are more sensitive to differences in sequences than others and the more sensitive ones are slower. Moreover, a single algorithm may have a parameter to control this sensitivity / time tradeoff.

```
*Query* =  pir|A01243|DXCH  232 Gene X protein - Chicken (fragment)
         (232 letters)
Sequences producing High-scoring Segment Pairs:             Score  P(N)      N
sp|P01013|OVAX_CHICK GENE X PROTEIN (OVALBUMIN-RELATED) (...  1191  7.7e-160  1
sp|P01014|OVAY_CHICK GENE Y PROTEIN (OVALBUMIN-RELATED).       949  7.0e-127  1
sp|P01012|OVAL_CHICK OVALBUMIN (PLAKALBUMIN).                  645  3.4e-100  2
sp|P19104|OVAL_COTJA OVALBUMIN.                                626  1.2e-96   2
sp|P05619|ILEU_HORSE LEUKOCYTE ELASTASE INHIBITOR (LEI).       216  3.7e-71   3
```

Figure 2.5: Sample output from a BLAST query. Shown are one-line descriptions of database sequences that match the query, including how closely they matched. Bioinformaticists identify biologically interesting properties using this tool.

Bioinformaticists explore biological hypotheses, searching among DNA fragments, using tools like BLAST [Altschul et al., 1990] and FASTA [Pearson and Lipman, 1988]. The BLAST nucleotide sequence similarity searcher from the National Center for Biotechnology Information is the most popular tool for this purpose. Sample execution output is shown in Figure 2.5.

Bioinformaticists share workstation farms — such as the dedicated 30 machines in the Phylogenomics Group of the University of California at Berkeley [Holliman, 2003] — and issue chains of fast, inaccurate searches to quickly demonstrate almost all non-matches followed by slow, accurate searches to confirm initial findings. Service time is dependent on the sizes of the sequences under comparison and how accurate the search is [Spring and Wolski, 1998]. Less sensitive searches take from tens of seconds to tens of minutes when matching against human genome sequences, while other searches can take up to six hours [Biowulf, 2004]. The accuracy v. runtime tradeoffs of different convergence algorithms can vary over several orders of magnitude.

A researcher is often able to plan a number of sequencing tasks ahead. In the extreme, some scientists wish to submit thousands of sequencing tasks [Biowulf, 2004] because they 'really do not know what [...] sequences will work.' [Giddings and Knudson, 2004] Batchactive scheduling (Chapter 5) would enable scientists to explore ambitious biological hypotheses without fear that resources would be wasted on speculative BLAST sequencing tasks that might be canceled after early outputs were scrutinized.

2.2.2 Sequential tasks

Another type of speculative work is a set of *sequential tasks* all from a single application performing the same function such as an any-time algorithm providing increasingly detailed outputs or ordered (temporal) outputs. The scenario for this type of speculative work concerns computer animation in which each task renders a movie frame.

Computer animated scenes are increasingly used in major motion pictures. The first full-length animated feature film created entirely by artists using computer tools and technology was Toy Story (1995) [Toy Story, 2004]. The films Finding Nemo, Shrek, Matrix [Taub, 2003], and Lord of the Rings [BBC News, 2004; Maya Association, 2004] have pushed the state of computer graphics technology. The following description of a computer-generated film's production's speculative nature I learned from speaking with Doug Epps from Tippett Studio and Tim Lokovic from Pixar Animation Studios [Epps, 2004; Lokovic, 2004].

Teams of hundreds of artists creating a computer-animated film at production houses such as Dreamworks or Pixar submit shots (scenes) for rendering, where each shot has roughly 200 frames, to a cluster of hundreds to thousands of processors. Each frame, which consists of up to 50 independent operations (for lighting, shading, animation, etc.) known as 'layers' can take minutes to hours to render. Shots are submitted using clustering tools like LSF [Platform, 2003] and proprietary tools like 'batchomatic.' Besides such software, Apple Computer, Inc. has recently developed a clustering solution explicitly for different aspects of computer graphics rendering called Qmaster [Think Secret, 2004].

For example, over a nine month production period, Weta Digital used $3,200$ processors to create Lord of the Rings: The Return of the King. This film had $1,400$ special effects shots, each containing at least 240 frames, and the average frame took 2 hours to render [Hillner, 2003]. A character from Lord of the Rings, including two intermediate layers, is shown in Figure 2.6.

This work is highly speculative. Artists submit a number of frames, up to a shot at once, for rendering. Upon seeing initial frames, an artist may decide that the lighting model is wrong, that a rendered object could be in a better location, etc. The overwhelming majority of computation never makes it into the final film [Epps, 2004; Lokovic, 2004]. Artists rarely get a scene right in one pass. They use rough renderings (successive refinement) to determine whether to continue or make changes.

The aggregate rendering operations for each frame could be considered a task, and the tasks to render a single shot could be considered a task set. The output of speculative tasks would often be desired in the natural order of the frames because artists often need to see successive frames to appreciate temporal characteristics, such as character motion. With a batchactive scheduler, aware of which tasks are speculative, the artist could prioritize key sections of a shot, those with more action, e.g., to more quickly decide

Figure 2.6: A completely computer-generated character (top) and two of its intermediate layers (bottom), designed by Weta Digital for Lord of the Rings. Source: Maya Association [2004].

whether additional frames are worth having. These frames would not only have priority above more speculative, remaining frames from that artist, but also from non-critical frames from other artists sharing the cluster. If it becomes known that unviewed output from speculative renderings will not be needed, the artist would cancel them so that they will not unusefully compete against other rendering tasks in the system.

2.2.3 Parameter studies

A *parameter study* [DeGroot, 1990] is a set of tasks exploring an often large parameter space. They usually begin by exploring the space in broad, shallow strokes, later to be refined to specific areas of interest. The scenarios for this type of speculative work are computer simulations.

Computer scientists routinely use clusters to submit chains of simulations exploring high dimensional spaces. Parameter studies for feature extraction, search, function optimization can continue indefinitely, homing in on areas for accuracy or randomly sampling points for coverage.

In the Electrical and Computer Engineering Department at Carnegie Mellon University, clusters are used by computer systems and computer architecture researchers for, among other things, studying microarchitecture cache behavior, computer virus propagation, and storage patterns related to I/O caching and file access relationships [ECE, 2002]. Many of these are discrete event simulations (sometimes trace-driven) which observe time-

```
Overall I/O System Total Requests handled:      10000
Overall I/O System Requests per second:         100.148471
Overall I/O System Completely idle time:        0.000000        0.000000
Overall I/O System Response time average:       49.917614
Overall I/O System Response time std.dev.:      8.392918
Overall I/O System Response time maximum:       81.359552
```

Figure 2.7: Sample output from a run of the DiskSim simulator. Shown are overall statistics from trace-driven I/O accesses to a simulated disk drive.

based behavior [Ball, 2004]. The service times for these tasks range from seconds to hours and are detailed in Table 4.4.

Simulation applications used by colleagues for parameter studies include SimpleScalar, DiskSim, and NS for researching microarchitecture, disk performance characteristics, and network performance, respectively [Simplescalar, 2004; DiskSim, 2004; NS, 2004]. The simulation results in this dissertation (Chapter 6.2) were created by extensive parameter studies (Chapter 6.1.3) using a simulator I wrote called `ba_sim` (Chapter 7.1) in which each hypothesis I explored consisted of tens to thousands of speculative `ba_sim` tasks per task set. In trying novel schedulers, simulations were canceled when the schedulers showed no significant difference in a random sampling of parameters. Sample output from the widely-used, accuracy-validated DiskSim tool is shown in Figure 2.7.

The Xfeed tool included in the Xgrid clustering software [Xgrid, 2004] from Apple Computer, Inc. explicitly supports parameter studies. One specifies a range of arguments (or a random sampling) to pass to a command. Xfeed generates task specifications for each possible combination of arguments and submits them; an example sweep would be through two dimensions of parameters in increments of 10 and 20, respectively.

With a batchactive scheduler, such simulations could occur in parallel with the experimenter analyzing desired and completed outputs and guiding the search in new directions, with speculative work — which will be canceled if determined to not be needed — operating in the background when pressing outputs from tasks among other users are needed.

2.2.4 Non-processor-based scenarios

While I focus on the processor resource (Chapter 3.3), for completeness I present examples of speculative tasks using the network and disk extensively. The network example can fit in an extended batchactive framework. Scheduling solutions for the disk cases take different approaches which I cite.

Web document prefetching has the strong potential to improve the experience of web browsing, a kind of exploratory search. Web cache hit rates are by their nature low (30–40% even with unlimited cache sizes), and thus web caches cannot by themselves eliminate web latency [Steere, 1997]. (Web page popularity follows the Zipf distribution:[1] while a small number of pages are exceedingly popular, the bulk see little reuse [Arlitt and Williamson, 1996].) A prefetching agent could construct a task set of prefetch candidates (perhaps by examining the links of the currently displayed web page). Web prefetches are speculative because they may or may not succeed in retrieving pages that the user is interested in viewing. User action (selecting links) may cause the prefetching agent to cancel prefetches and issue new ones. A batchactive scheduling policy would determine how many such prefetch tasks (network accesses) to issue which would be *in moto* while the user reads the previously desired web page (i.e., during the user's think time). The time to retrieve a page and think time for a user browsing the web are both Pareto distributed [Crovella and Bestavros, 1995] with expected values under ten seconds. The scheduler would attempt to balance the response time that the person experiences with the fractional increase in network usage caused by prefetching. It is in the user's interest to control network usage because the network might also be used for demand-driven work and because network usage might cost on a per-byte basis (e.g., some low-bandwidth wireless connections). A more detailed discussion of scheduling speculative network requests is presented in Chapter 5.9.

Data mining is often an exploratory search, with the relevance of future queries dependent on recent outputs. The size and prevalence of data processing workloads has grown enormously [Fayyad, 1998], making such mining expensive in time and cost. Increasingly, content is not bound by particular choices of data organization and the delivery of information is in forms that go beyond traditional list management and database report methods [Sculley, 1989]. The Diamond system searches 'loosely-structured data' (AutoCAD drawings, USGS maps, CAT scans, etc.) more efficiently by quickly discarding unneeded information at the data source [Huston et al., 2003], a type of task cancellation called 'throttling' in DeGroot [1990].

Disk I/O often limits application performance. To the extent that an application knows its future data needs, its performance can improve by disclosing these accesses before the application stalls for unread data. These

[1] The Zipf distribution [Zipf, 1932] is related [Crovella, 2000] to the Pareto distribution (Chapter 6.1.3), sharing its heavy-tailed property. George Zipf discovered that the probability of encountering the rth most common word in a corpus is roughly $P(r) = 0.1/r$ for r up to about 1000 [Weisstein, 2004i].

disclosures are speculative because the data within early reads may determine which future reads are actually needed. This is the concept behind the TIP system [Patterson et al., 1995] in which programmers manually disclose future data needs, covered in more detail below (Chapter 2.5.3).

2.2.5 Summary of scenarios

Speculative work and speculative scenarios cover broad areas, are important, and are becoming more common; the speculative work patterns of Chapter 2.1 occur across many types of users executing many types of applications. Scientists, researchers, private individuals at commercial ventures, academic institutions, research laboratories simulate phenomena and evaluate hypotheses by issuing tasks speculatively.

I described three types of speculative work in this section. An exploratory search is a hand-crafted, based on domain-specific expertise, collection of tasks to confirm or refute a hypothesis. I cited the usage of BLAST in the field of bioinformatics as an important scenario. Sequential tasks perform a single function in which the order that outputs are desired is well-defined. The important scenario for this type of speculation is rendering the frames of a computer-animated film. Parameter studies begin as broad, shallow explorations of a high-dimensional space that are refined to areas of interest. The important scenario here includes any type of parameterized computer simulation. I also discussed non-processor-based speculative scenarios, including data mining and web prefetching.

Users submit work they know they need and work they do not know they need (speculative work) and traditional schedulers do not have the knowledge to treat these types of tasks differently. The batchactive approach to scheduling speculative tasks (Chapter 5) improves time- and cost-based metrics for these scenarios by intelligently prioritizing among needed and speculative tasks between one user and among all users sharing a computational resource. Ambitious hypotheses potentially requiring an unbounded amount of resources can be explored without fear that resources would be wasted on long-shot speculation. All that is required is for users to disclose their speculative plans as task sets and then request individual task outputs when it becomes known that they are needed. I do not believe the cost for users to generate speculative task sets in the above scenarios is large.

2.3 Enabling behavioral conditions

Besides the behavior of users and the kinds of application they run described above (Chapters 2.1 and 2.2), there exist several user behavioral conditions suited to batchactive scheduling.

Speculative work is often desired some time after the work is submitted. This occurs for two reasons. The first is that a user needs time to think about (viz., the 'think time') the most recently received task output before being ready to consume the subsequent task output. The second is that a user may become temporarily unavailable to process new task output independent of the nature or availability of previous task output, such as when the user leaves the office for the day and will not resume work until the next morning. I also describe the prevalence of idle computational resources that can be leveraged for speculation and argue that even apparently busy servers can fruitfully engage in speculation.

2.3.1 Existence of think times

The speculative systems covered in related work (Chapter 2.5) share the property that think times exist and that think times are leveraged to improve performance. However, the notion of think time — one element of a system not being ready to consume output from another part — is not always given this name, and does not necessarily involve a human. For example, in the TIP prefetching system [Patterson et al., 1995], an instruction stream consumes prefetched data. The time that the processor is executing instructions (i.e., not stalled on data) is effectively its think time during which the (potentially speculative) data prefetching can be pipelined. In hardware instruction speculation [Hennessy et al., 2002], the delays incurred by the memory subsystem is effectively think time during which the speculative execution of instructions can be pipelined.

The existence of user think times provides the opportunity for a speculative scheduler to choose a task ordering that better (Chapter 6.2.2) meets user and resource provider scheduling goals (Chapters 4.4 and 5.3) because a user does not need the outputs of every speculative task at once; the user is either 'blocked on' one task's output or 'thinking about' the output of the previously received task output (Figure 1.1). Think time gives a batchactive scheduler the flexibility to delay the execution of non-pressing, speculative tasks in deference to known or more likely to be needed tasks. Since speculative tasks might be canceled, delaying their execution might even result in eventually unneeded tasks being canceled before they consume significant, if any, resources. This section cites situations in which think times were measured. The non-obvious dependence on think time for batchactive improvements is confirmed in experiment (Chapter 6.2.4).

Bubenik and Zwaenepoel [1989] built a system to optimistically build (compile and link) software applications before the developer explicitly asks to do so (Chapter 2.5.1). Part of their work measured the time between a build being needed and the last time a source file that is part of the build was modified. This 'out-of-date time' is a conservative estimate of think time because there was additional time that the developer spent modifying source files not included. 'Out-of-date' time is important to them because source files need to be saved before builds can proceed speculatively.

They observed that most build targets are requested soon after a change to source files but that these rebuilds usually consist of compiling one source file; a developer testing a small change, e.g. However, sometimes users wait a long time between source modification and executing a build request, and these builds are more likely to involve more computational work. Specifically, the median out-of-date time was 32 seconds while the mean was 378 seconds.

In the domain of web browsing, think time also exists. Many web prefetching schemes rely on think time to reduce browsing response time [Steere, 1997; Padmanabhan and Mogul, 1996; Bestavros, 1996; Fisher, 2002]. Crovella and Bestavros [1995] found that web think times (also called 'off-times') are Pareto distributed, a discovery used in their accurate web workload generation tool called Surge [Barford and Crovella, 1998].

2.3.2 Existence of away periods

This section describes a related concept to think time that further aids batchactive scheduling. A user can have *away periods* during which the user does not wait for task output. People routinely become unavailable to consume task output and knowing when this occurs enables a scheduler to better order work, especially if the scheduler knows which tasks are speculative. Away periods can be thought of as the creation of think time independent of task completion — e.g., a user leaving at the end of the work day and not being ready for output until the next morning (Figure 1.1). (The prediction of away periods was used in other work to decide when to restore the memory state of cluster desktop workstations harnessed for remote execution [Petrou et al., 1996].)

To help see the flexibility provided by knowing away periods, consider the following example taken from Feitelson et al. [1997] and illustrated in Figure 2.8: 'Assume that a task needs 3 hours of computation time. If the

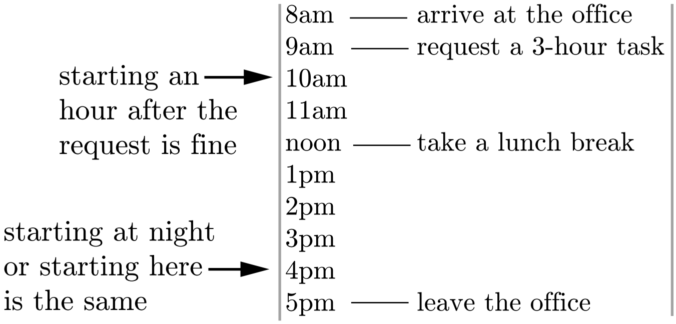

8am — arrive at the office
9am — request a 3-hour task
10am
11am
noon — take a lunch break
1pm
2pm
3pm
4pm
5pm — leave the office
starting an hour after the request is fine
starting at night or starting here is the same

Figure 2.8: How knowing away periods gives a batchactive scheduler opportunities to delay some speculative tasks so that more pressing tasks can run.

user submits the task at 9am, he may expect to receive the results after lunch. It does not matter to him whether the task is started immediately or delayed as long as it is done by 1pm. Any delay beyond 1pm will reduce user satisfaction. However, if the task is not completed before 5pm, it may be sufficient if the user gets his or her results early next morning.' Further, because my application scenarios are speculative, tasks delayed because the submitting user was in an away period that were later determined to not be needed would not have competed for potentially scarce resources.

Because of the difficulties in knowing away periods (Chapter 5.7), the policies, results, and conclusions of this thesis do not rely on predicting or obtaining away periods. I consider away periods an additional opportunity beyond think time for a batchactive scheduler to more effectively execute potentially needed work for even better performance.

2.3.3 Existence of server idle time

Batchactive schedulers leverage spare computational resources for executing speculative tasks. A batchactive scheduler that improves performance often increases load with speculative tasks as a side-effect (Chapter 6.2.2).

Cluster workstations are available 60–80% of the time [Mutka and Livny, 1987; Douglis and Ousterhout, 1991; Arpaci et al., 1995; Acharya et al., 1997]. Arpaci et al. [1995] state that, 'although the set of idle machines changes over time, the total number of idle machines stays relatively constant [. . .] even during the busiest time[s].' Many additional citations stating similar statistics can be found within the above studies. Even the highly desired resources at the Pittsburgh Supercomputer Center (PSC) are idle 10% of the time [Harchol-Balter, 2003a].

Note, however, that such idle time statistics are a conservative estimate of the resources available for speculation. While, anecdotally, some report that during 'crunch times,' resources are saturated [Epps, 2004; Lokovic, 2004], what is important for improving (Chapter 6.2.2) scheduling metrics (Chapter 5.2) is the load made up of needed tasks; i.e., the total load minus the load made up of speculative tasks. This is because such speculative load can be delayed until needed or canceled. When an interface exists for and is used by the user to distinguish among needed and speculative tasks, it may become apparent that much of the total load comprises speculative work that can be scheduling more effectively with a batchactive scheduler.

2.4 Common practice and its deficiencies

Non-speculative schedulers (Chapter 4) were not designed for and poorly support speculative workloads. Yet users regularly submit batches of speculative tasks as shown by the scenarios above (Chapter 2.2). These task sets mix with known needed tasks (Chapter 4.6), confusing the scheduling policy's attempt to meet scheduling goals. Further, because speculative tasks look like needed tasks, when resources are charged, they are charged regardless of whether their computations were eventually needed (Chapter 4.2).

Should a user with speculative tasks submit one speculative task, a few, many, or the entire task set? The user wishes to reduce the time he or she waits for needed task output and the user wishes to reduce how much he or she will be charged for unneeded speculation. There is confusion as to how many tasks a user should submit, resulting in ineffective scheduling; i.e., poor time- and cost-based scheduling metrics (Chapter 5.2). When resources are not directly charged (such as in a communal cost-center) or if all the users have the economic resources to pay for wasted work, then the resources will be overwhelmed, making the system unusable.

Further, traditional metrics are insufficient when users behave speculatively and have think times and away periods. According to Feitelson et al. [1997], 'The use of metrics such as throughput and response time [...] may be due to the simplicity of the evaluation, or it may be a sign of some non-obvious influence from theory.' What is more important is the response time experienced by the user, the *visible response time* of needed, no longer or never initially speculative tasks, which accrues outside of think time and away periods, introduced in Chapter 5.2. Relevant metrics (Chapter 5.2), a new pricing mechanism (Chapter 5.1), and new speculative policies (Chapters 5.4 and 5.5) overcome (Chapter 6.2.2) the deficiencies of existing cluster scheduling.

2.5 Related speculative work

Speculation to improve performance is found at the level of I/O requests, program blocks, and instructions across all areas of computing including architecture, languages, and systems. I discuss related work for scheduling across speculative tasks such as those scenarios discussed above (Chapter 2.2), scheduling speculative parts of a single task, and scheduling speculative activity that uses resources other than the processor. The latter two types of speculation, which are outside my scope (Chapter 3), are covered because they inform my terminology and solutions (Chapter 5) and they place my solutions in context.

When speculation has time or space overheads, one tradeoff is between unbounded speculation and delayed evaluation [Wikipedia, 2004] (related to lazy evaluation and described in Chapter 1). This tradeoff is found most often in the context of functional programming and appears in some of the systems below. Another reoccuring issue is how sometimes unneeded speculation must be prevented from affecting other state.

2.5.1 Speculation across tasks

Bubenik and Zwaenepoel modeled a cluster of users engaged in software development using a modified `make` tool [Bubenik and Zwaenepoel, 1989]. At each save of a source code file, their system speculatively runs the compiler using the build rules encoded in the project's `Makefile`. Their seminal work measured the potential to reduce visible response times from building applications speculatively. Their simulator modeled one task (rebuild) pending per user. My model is broader, encompassing users who operate interactively or who submit batches of speculative work for a number of scenarios (Chapter 2.2), including users behaving speculatively with non-speculative schedulers (Chapter 2.1). Their work isolates speculative compilations from the rest of the system. This is not needed in my system which stores speculative outputs in isolated locations until requested. Beyond their study of time-based metrics, I also study resource cost as it relates to user charges and server revenue.

The Xcode integrated software development environment for Apple computer architectures has a predictive compilation [Xcode, 2004] feature which begins file compilation even while files are being edited. This single machine, single user speculation does not address scheduling issues. The Xcode documentation advises turning off predictive compilation on slower machines when it may interfere with other activity including the editing itself.

In the database realm, Polyzotis et al. built a speculator that begins work on database queries, where each query could be considered a task, during the user think time in constructing complex queries [Polyzotis and Ioannidis, 2003]. They use machine learning techniques to predict what the user will need before the query is finished, but they do not consider the scheduling issues of a competing set of users submitting needed and speculative queries.

Eggert's research on speculative scheduling examined how idle resources could be leveraged for background or speculative work [Eggert, 2004]. Due to preemption costs, he quantifies situations in which it is beneficial for the scheduler to be non-work-conserving (i.e., to idle when work is ready to run). This research does not consider the scheduling issues that arise with speculative chains of tasks or contention from multiple users.

Sun et al. [1999] introduce a 'parallel world' in which each set of speculative tasks receives a private execution environment that is merged into the real environment when speculative outputs are needed. They address how to coherently integrate or discard requests to modify file system state from speculative tasks depending on whether these tasks where eventually determined to be needed or not, respectively. This work does not explore the scheduling issues of speculative tasks. My work does not have the need for private execution environments or encapsulations because the outputs of scientific tasks or experiments are stored in locations that do not interfere with the operation of the system. Outputs are provided to the user when requested, and discarded when not needed.

2.5.2 Speculation within tasks

At the hardware level, speculation is commonly used to improve performance when one part of the architecture presents a bottleneck to another. For example, instructions are executed speculatively when control reaches a branch and the resolution of the branch depends on data from a slower part of the memory hierarchy. Without this speculative execution, the processor would stall until the data became available. Because speculation may be incorrect, its effects must be isolated until resolution. [Hennessy et al., 2002]

Osborne [1990] describes how Multilisp [Halstead, Jr., 1985], a version of Scheme [Abelson and Sussman, 1996] with parallelism constructs, can be used for speculative execution. Their examples include parallel search, a parallel `if` statement (which works on both the 'consequent' and 'alternate' branches), among others. The difference between this work and others within computer languages is the grain of computation: when small, the concerns of speculative overhead and isolation are greater; when large, the concerns shift to the scheduling of multiple tasks from competing users exhibiting a range of behaviors over larger time scales.

Other work can transform a single executable into speculative pieces, although sometimes programmer annotations are required [Bubenik and Zwaenepoel, 1990; Cowan and Lutfiyya, 1995].

The bandwidth-delay product of current and future grids have spurred speculative approaches to improving the performance of tightly-coupled applications [Chrisochoides et al., 2003; Lee, 2002]. Such work examines how to rollback unneeded computation within an application and throttle work so that speculation does not overly consume resources. Rollbacks for optimistic computing derive from the virtual time concept [Jefferson, 1985]. In contrast, I speculate among multiple independent (Chapter 3.2) tasks and I study how to schedule among task sets from multiple users.

2.5.3 Speculation on non-processor resources

Patterson et al. [1995] have shown in the TIP system how application performance can increase if the application discloses storage reads in advance of when data is needed. Programmers insert speculative data reads as program annotations in the hope that the system can use this information to reduce application I/O latency. My work applies the same concepts and terminology to the processor resource at the granularity of tasks. While their work focused on storage questions, such as how to balance cache space between prefetches and LRU caches, because of the relative size of memory and (potential and known) data demands, I assume that sufficient storage exists to store speculative task output; if this is not the case, speculative execution can be throttled or speculative outputs can be dropped, lessening the benefit of batchactive scheduling. As TIP uses disclosed reads to exploit storage parallelism, batchactive scheduling uses disclosed tasks to exploit cluster parallelism. TIP gives priority to demand requests. I introduce a spectrum of batchactive schedulers, but the implemented policies (Chapter 5.5.1) also share this property of preferentially scheduling known-needed tasks. When task speculation costs (e.g., time, space overhead, or monetary costs), I contrast a feedback-based technique applicable in my dynamic environment to the analytic-based technique used by TIP (Chapter 5.9.3).

An extension to TIP by Chang and Gibson [1999] automatically discloses I/O accesses by optimistically running sandboxed copies of an executable and monitoring its accesses.

In the approach of Steere [1997], users disclose sets of data objects called 'dynamic sets' that they might need in which the expected order of desire is unknown or irrelevant. Calls to request data may return objects in any order, giving the I/O system, whether using the disk or network, the flexibility to re-order accesses for better performance. E.g., cached objects may be returned first while prefetching proceeds for other objects, possibly in parallel.

Researchers have sought to reduce network delays by discriminating between speculative and requested network transmissions. Padmanabhan et al. have shown a tradeoff in visible response time and fractional increase in network usage when varying the depth of their web prefetcher [Padmanabhan and Mogul, 1996]. (Prefetch candidates are determined by server-inserted annotations into web pages. An Internet standard for such annotations exists, RFC 2068, Section 19.6.2.4, enabling stock browsers to optionally act on these annotations. A web server could predict what pages a person might request by analyzing past access patterns using, e.g., a technique based on Markov models [Deshpande and Karypis, 2000].) In the approach of Steere [1997], people construct sets of web prefetch candidates and the browser prefetches as much as three such candidates simultaneously until all are fetched, or until the person initiates new activity. (They argue that candidates should be manually constructed to maximize the potential of a prefetch being used.) The Mozilla web browser will download candidate documents (based on server-provided annotations) after the requested page has loaded and will stop when there is nothing left to prefetch or when the person selects a link [Fisher, 2002]. TCP Nice consists of sender-side changes to TCP congestion control to enable low-priority network service that could be used for background or speculative network accesses [Venkataramani et al., 2002].

2.6 Summary

While there are no significant situations in which batchactive scheduling performs worse than a non-speculative scheduler, there are situations that are more applicable to batchactive scheduling than others. This chapter described applications that lend themselves to speculation and related work on speculative systems.

Users routinely run tasks speculatively. A speculative task is one whose output is not known to be needed at the time of submission. I showed that it is common for users to submit speculative task sets, which could vary between a few and thousands of tasks, and cycle between thinking about completed task outputs and waiting for task outputs that they come to know are needed; and that this behavior will become even more common due to user time costs and resource cost trends. Task sets are usually constructed by the user using domain-specific, expert knowledge. Other times an autonomous agent working on behalf of a user can construct large task sets automatically; this is easier when the order in which tasks complete is irrelevant.

I classified speculative work into three categories. Exploratory searches consist of hand-crafted task sets to answer some scientific hypothesis. The example I gave was tasks that sequence DNA. Sequential tasks consists of the same task run many times, producing outputs viewed in sequence. Here I described the rendering of computer-animated films, in which each task renders a frame. Parameter studies are searches of high-dimensional spaces, often sampling the space broadly and later focusing on areas of interest. Computer simulations of a broad variety of phenomena, each run being a data point-producing task, was the motivating application for this type of speculative work. The application examples I gave all conform to the limitations I place on the scope of this thesis (Chapter 3). I also discussed non-processor-based use of speculation.

The benefits of batchactive scheduling are related to when users need speculative work and the available server idle time. I discussed the existence of user think times, user way periods, and server idle time to show that batchactive scheduling is applicable in today's clusters and grids. Trends suggest increasing applicability.

Finally, I discussed problems with the common practice of submitting speculative work to non-speculative schedulers and I described existing systems that apply speculation, contrasting their approaches with mine for the cases where our goals were most similar.

> When a man's tied to a wheel and that wheel turns, he turns.
>
> Mike Watt, *Burstedman*

3 Scope

I restrict the scope of my work to avoid issues orthogonal to my thesis that a speculation-aware scheduler can order tasks to provide, among other goals, better visible response time at lower resource usage. These restrictions concern the target application domain, computer architecture, and computer resource. The scheduling solutions I introduce, even with these restrictions, supports compelling and wide-ranging real computational scenarios faced by researchers, scientists, and others with heavy computational demands, including those described in Chapter 2.2. Further, areas of computation in which the scheduling solutions I present do not directly apply can often be supported with straightforward extensions to batchactive scheduling. I mention such extensions below but I do not explore them.

3.1 Target application domain

For my work to provide an improvement over common schedulers, there must be speculative tasks. Users, or autonomous agents working on their behalf, must be able to disclose some potentially needed tasks before requesting their output. Evidence that people can plan ahead in this manner was described in the work patterns and application scenario sections of the previous chapter (Chapters 2.1 and 2.2). Scientists in national laboratories, academic institutions, and private research departments can often construct series of experiments consisting of speculative tasks that could take a wide range of computing time, sometimes an indefinite amount of time. What remains is for users to disclose their computational plans, i.e., to identify which tasks are speculative by first disclosing them and later requesting them. I believe that this additional work by the user required by my scheduling solutions is not a large burden, especially when motivated by the improvements in response time and resource usage that my results in Chapter 6.2 demonstrate, and is easier and more effective than the alternative of users independently throttling the submission of speculative tasks to non-speculative schedulers.

I assume that there exists sufficient storage to hold speculative task outputs; that the cost for a server to hold speculative outputs indefinitely is insignificant. If this is not a case, speculative execution must be throttled or speculative outputs must be dropped, lessening the benefit of batchactive scheduling.

For schedulers that rely on knowing task size, such as those that employ SRPT, task service time (the computational demands of a task in cycles or processor time) should be predictable with some precision. I have evidence that they are which I present later (Chapter 4.7). In any case, my batchactive scheduling solutions, as my results in Chapter 6.2 show, do not depend on underlying schedulers that require task size. Almost identical performance improvements are gained by comparing a batchactive policy based on FCFS with a standard, non-speculative FCFS scheduler, e.g.

I also assume that the execution of a task affects no other tasks or state other than the output it produces which the system stores in an isolated location until delivered to the user. Thus, there is no consequence of running a disclosed but ultimately unneeded task in the sense of requiring state rollback. This property is shared by the processor-bound applications motivating this research (Chapter 2.2). Other situations [Sun et al., 1999; Chrisochoides et al., 2003; Chang, 2001], elaborated on in Chapter 2.5, require rollback mechanisms.

Applications with real-time requirements are outside my scope. None of the important application scenarios in Chapter 2.2 have real-time requirements. Batchactive schedulers, built on non-real-time or modified non-real-time schedulers, are best-effort and do not guarantee when tasks will complete. As a loose analogy, I term the time that a user needs the output of a previously disclosed speculative task to be the task's *deadline.* Taking this analogy further, one could phrase the visible response time goal of batchactive scheduling as optimizing a soft real-time utility function defined as the total amount of time these deadlines were overrun. However, in contrast to real-time work, batchactive schedulers do not need to know when these deadlines might be, i.e., when a user will need task output (if the user will need speculative task output at all). More comparisons of my work against real-time scheduling may be found in Chapter 5.6.5.

3.2 Target architecture

I focus on the cluster (networks of workstations or distributed server systems), whose form is shown in Figure 1.4, as it is a cost-effective and flexible architecture for building small computing resources to supercomputers to computational grids. For small to medium-sized computing resources, clusters are pervasive. Clusters comprise 60% of the top 500 supercomputers, with the MPP (massively parallel processor) being the second-most common architecture [Top500, 2004]. Clusters also form one of the computational building blocks of grids [Berman et al., 2003; Foster and Kesselman, 2004], a meta-architecture of potentially heterogeneous and widely distributed computational resources introduced in the mid-1990s. More discussion on the cluster architecture is found in Chapter 4.1.

Clusters are typically loosely-coupled, in the sense that communication latencies (and latency variance) among nodes are higher than within one SMP (symmetric multiprocessor) or an MPP which employ specialized networking hardware among processing units. Thus, relative to other architectures, applications suited to clusters either do not communicate or do not communicate at fine-granularity.

In contrast, there exist 'space-shared,' parallel applications (SPMD or MPMD) which require reserving a number of nodes toward executing a single application. They communicate frequently and are more suited to tightly-coupled architectures. Scheduling goals for resource providers supporting such tasks are to provide a high load (utilization) of a resource's processors without unduly starving any task. For non-speculative tasks, there are many scheduling approaches for different goals and applications with different requirements (such as high preemption costs). One approach is gang-scheduling (coscheduling) [Black, 1990]. The fewer the application requirements, the more orthogonal such solutions are and the easier they can be combined with my speculative scheduling policies.

Parallel applications are outside my scope. Instead, I focus on non-communicating, timesharing applications in which each task uses one processor. (On my system, a parallel application requiring n nodes could be considered one large task that would not start until n nodes became available and would not end until the last node finished its work. The n nodes would be considered one large processor.) The number of important computational problems that consist of non-communicating tasks is large, as evidenced by the application scenarios in Chapter 2.2; thus the scheduling solutions I provide are highly applicable to existing problems even when not supporting space-shared tasks.

I study both resources that are not directly charged (such as when users cooperate toward a shared goal in a laboratory setting) and resources that are charged (such as third-party outsourcing). This difference has implications on scheduling goals related to user costs and resource owner revenue and the prevention of resource abuse described in Chapter 4.2.

3.3 Focus on the processor resource

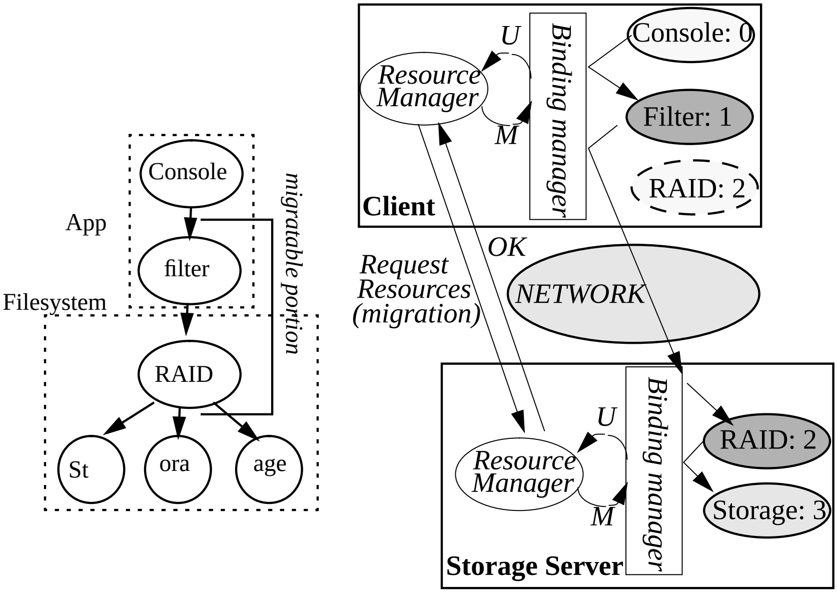
Because the processor is often the primary bottleneck for the important scenarios listed in Chapter 2.2, this thesis addresses speculative scheduling for the processor resource only. Scheduling other resources, such as disk, network, memory, and energy, is outside my scope.

To focus on how speculative tasks should be scheduled differently from non-speculative tasks, I assume that a task only uses the processor significantly. Tasks that block on other resources significantly will not benefit as much from more intelligent processor scheduling. In general, a task may use multiple resources sequentially or in parallel. For example, a processor-intensive task might use the disk to load a dataset. But I assume that those disk accesses are an insignificant part of the task's work. While some motivating applications in Chapter 2.2 have large data requirements, I/O is typically not the bottleneck.

Jim Gray (winner of the 1998 ACM Turing Award) refers to such applications as fitting well in clusters because of the cheap availability of fast I/O at the scale of the LAN [Gray, 2004]. Many solutions exist to address demanding application I/O requests, including a novel high-performance architecture developed in the Parallel Data Laboratory at Carnegie Mellon University called NASD that separates the I/O control and data paths [Gibson et al., 1998]. A recent IETF Internet-Draft [Gibson and Corbett, 2004] culls this and similar technologies for the proposed development of a scalable file system called pNFS [Hildebrand and Honeyman, 2004]. When needing to support applications, beyond my scope, in which the cost of moving data is high, Gray [2004] recently suggested a long-standing idea independently explored by a colleague, Khalil Amiri, and I for automatically migrating pieces of computation to the location of the data. Our system is called Abacus [Amiri et al., 2000] (Figure 3.1), and I advocate its approach as an extension to the batchactive scheduling of this thesis for situations in which I/O is the bottleneck.

Moreover, a task, whether its processing is speculative or not, may use other resources speculatively. (Existing work on supporting speculative use of non-processor resources is listed with related work in Chapter 2.5.3.) My solution to scheduling speculative tasks on the processor does not preclude applying solutions for speculative use of other resources to get additional benefits, although interactions among speculative systems is interesting future work. Should a non-speculative disk access of a speculative task have more or less priority than a speculative disk access of a non-speculative task?

For scheduling policies that preempt tasks, I assume that preemption



(a) Abacus modules

(b) Abacus components

Figure 3.1: Overview of the Abacus module migration system. This system, which I co-designed, may help alleviate the I/O bottleneck for data-intensive cluster applications, putting them in the scope of batchactive scheduling solutions. Data-intensive applications are partitioned into modules that can independently migrate between client computers (workstations, consoles) and the storage servers holding the data they act on. Shown on the right are the principal components of the Abacus system and on the left is an example application which mines data stored in a striped file. This application consists of migratable filter and RAID modules (console and storage modules are fixed at the client and server). Migrations are intended to improve the roundtrip latency of a data access beginning at the console module in response to dynamic conditions such as server load, network load, and filter selectivity (the amount of data produced per data consumed) — i.e., the system can switch at the granularity of a module between being function-shipping and data-shipping. Inter-module calls are transparently redirected by binding managers and statistics concerning these invocations are monitored by resource managers (indicated by 'U'). Resource managers across servers cooperatively make migration decisions to improve global performance, asking binding managers to enact module migrations (indicated by 'M').

costs are low. These policies have not been tailored to tasks with out-of-core memory requirements, e.g., although I believe that adapting known techniques would be straightforward. Scheduling literature shows how various techniques such as backfilling and long-quanta preemption[1] can meet these goals [Feitelson and Jette, 1997]. I believe that such techniques would equally affect standard and speculative processor schedulers, thus I factor out such interactions in my studies.

Further, I assume no complex interactions or dependencies among tasks, such as tasks that contend for shared locks. Such interactions, it is my belief, would equally frustrate the batchactive and non-speculative schedulers under study.

3.4 Summary

This chapter restricted the scope of this thesis, enabling me to focus on how a scheduler should schedule speculative and non-speculative tasks while avoiding issues extraneous to this question. The strongest assumption that I make, and have justified in Chapter 2, is that there exists a domain of applications consisting of speculative work, and that in this domain, users or agents can take the effort to disclose their computational plans. I limit my architecture to the cluster, which is the most important architecture for high-end computing. I study two prevalent economic relationships between resource provider and user: a resource provider selling cycles to a user and a dedicated resource whose usage is not directly charged to the user. Finally, I focus on the processor resource which is the bottleneck for the many important scenarios described in Chapter 2.2. I make some additional assumptions on resource usage, such as low preemption cost. For applications for which these assumptions do not hold, there exist traditional, orthogonal scheduling solutions for non-speculative tasks, and references were supplied. While some could be applied to batchactive scheduling, I have not done so.

[1] Non-preemptive scheduling or scheduling with long (ten minute) quanta are used because many supercomputer applications take up so much memory that only one can fit in core at once [Harchol-Balter, 2002].

> [N]one of us could get very far in discovering any part whatever of the Truth if we could not make trains of reasoning [. . .] as nearly mechanical as possible.
>
> Philip Jourdain, *The Nature of Mathematics*

4 Non-speculative scheduling

In this chapter I discuss non-speculative scheduling: traditional, existing scheduling that does not know which tasks are speculative and thus cannot treat them differently from non-speculative tasks. Although, from a user's perspective, submitted tasks may be speculative whether or not the scheduler supports speculative tasks, speculating with non-speculative schedulers results in a mismatch of scheduling goals. This chapter provides the terminology and background necessary to discuss batchactive scheduling (Chapter 5), developing a foundation to clarify how batchactive scheduling benefits speculative user behavior.

The organization of the primary entities under discussion is depicted in Figure 4.1. Self-interested users (or agents working on their behalf) interact with clustering software. Any number of users request and cancel one or more tasks. Cancelation occurs when a user had thought that he or she needed a task result but then reconsidered. This speculation is common on standard schedulers, and I later show that better scheduling can result from the batchactive schedulers (Chapter 5) that distinguish between tasks that are more or less known to be needed. A scheduling policy decides which and when requested, non-canceled tasks run. The scheduler communicates decisions to the operating systems running on the cluster resources which handles the details of running tasks on the servers (such as forking processes) and provides task statistics (such as resource usage) to the policy. The policy may use these statistics to try and make better scheduling choices in the future. After a task executes, its output is supplied to the user that requested the task.

I begin by describing the target architecture for the scheduling approaches in this thesis. Then I describe extant pricing mechanisms for this architecture. Following I concretize the interaction of servers, tasks, and users with formal definitions and metrics. Building on these metrics, I introduce scheduling goals, both from the user's perspective and from the resource provider's perspective. With these goals in mind I first describe scheduling

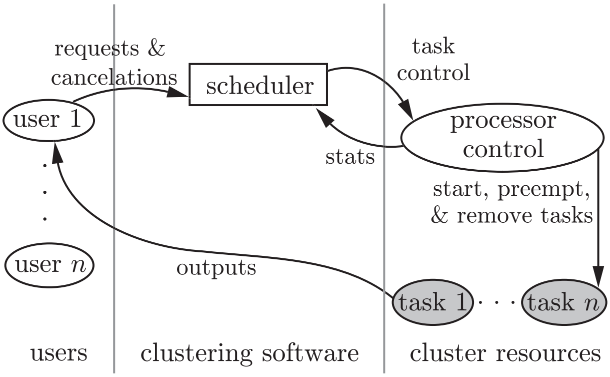


Figure 4.1: Interaction between users, non-speculative clustering software, and cluster resources. (Compare to the speculative version depicted in Figure 5.1.)

policies in theory and then in practice. Since some policies require knowledge of task service time (the time it takes for a task to run on an otherwise unloaded system), I also speak about ways to obtain this value. The final section of this chapter describes how non-speculative scheduling poorly achieves important scheduling goals when speculative tasks exist.

4.1 Architecture

Both the non-speculative scheduling of this chapter and the speculative scheduling of Chapter 5 apply to the computing architecture described here.

I focus on the cluster as it is a cost-effective and flexible architecture for building small computing resources to supercomputers to computational grids (Figure 1.4). The resource[1] of concern is processor time (Chapter 3.3). Clusters are also known as networks of workstations and distributed server systems. Cluster nodes may be single to several processor machines, such as an SMP. Collections of multiprocessors are common in high-performance computing [Schroeder and Harchol-Balter, 2000]. Clusters provide numerous advantages over the MPP architecture, including the effect of volume manufacturing on price / performance, scalability, fault-tolerance, and incremental upgrade. [Anderson et al., 1995; Pfister, 1995] Beowulf clusters, with their focus on dedicated, homogeneous, inexpensive yet high-performance commercial, mass-produced commodity hardware and open source software

[1] A resource is an instrument to create goods or services that is both scarce and useful; i.e., having a finite supply and non-zero demand [Baumol and Blinder, 1994] [Narayanan, 2002, ch. 2].

have become a common architecture over the last decade for high performance computing [Beowulf, 2003].

For small to medium-sized computing resources, clusters are pervasive. The Parallel Data Laboratory research group, to which I have belonged during the development of this thesis, has a cluster comprised of tens of machines, some under the control of the Condor clustering system [Condor, 2003], dedicated as cycle servers for experiments and simulations. Other groups within the Electrical and Computer Engineering department employ clusters. The computer architecture group uses Condor to distribute Simplescalar [2004] microarchitecture simulations across a cluster [Wenisch, 2003]. Two graduate courses within this department also run tasks on this heavily used cluster. Other departments within Carnegie Mellon University (such as the physics and computer science departments) have clusters for other computational purposes. The machine learning group has a cluster of twelve machines for neuroscience research, analyzing three-dimension grids of FMRI data and training classifiers for brain states; their approach to running tasks is less sophisticated, with users manually logging into machines to start tasks [Pereira, 2003]. These application examples are single-node, non-communicating tasks.

The concept has existed for over a decade at other universities, governmental laboratories, and commercial entities: Scientists share a workstation farm of thirty machines at the Phylogenomics Group of the University of California at Berkeley to study biological hypothesis [Holliman, 2003]. Computer graphics artists at Weta Digital use a cluster of 3,200 processors to create films such as Lord of the Rings [Hillner, 2003].

Clusters comprise 60% of the top 500 supercomputers, with the MPP (massively parallel processor) being the second-most common computer architecture [Top500, 2004]. Supercomputer centers favor clusters for their ease of administration, scalability, and price [Schroeder and Harchol-Balter, 2000]. Moreover, scientists try to do as much work as possible on clusters because the programming environment on a supercomputer is restricting: there are many assumptions such as on memory usage that makes code non-portable once designed for a particular supercomputer [Lopez, 2002]. The Virginia Tech cluster of 1,100 Apple G5 Power Macs is an example of a supercomputer built from a cluster [Virginia Tech, 2004].

Clusters also form a computational building block of grids [Berman et al., 2003; Foster and Kesselman, 2004], a meta-architecture of potentially heterogeneous and widely distributed computational resources introduced in the mid-1990s. The grid problem is the controlled and coordinated resource sharing and resource use in dynamic, scalable virtual organizations [Foster et al., 2001]. A grid provider has the infrastructure for accounts, authentication, code portability, and resource discovery. Grid computing is a vast commercial and research initiative spanning various computer architectures and granularities of computation that is predicted to grow increasingly important in the near future.

Cluster (and grid) resources are provisioned by freely available or commercial clustering software such as Condor, Xgrid, Platform LSF, the Globus Toolkit, PVM, Legion (Avaki), and the Sun ONE Grid Engine [Condor, 2003; Xgrid, 2004; Platform, 2003; Globus, 2003; PVM, 2004; Legion, 2004; Sun Grid, 2003].[2] The stock schedulers of these software systems can be replaced or extended with varying degrees of ease. The scheduling solutions in this thesis can be deployed by replacing or extending the scheduler in existing clustering software with a scheduler for speculative tasks as described in Chapter 7.2.

The restrictions in scope that I made to focus on important cluster-based software was described in Chapter 3.2. To recapitulate, I focus on single-node, non-communicating tasks, of which there are numerous, important examples (such as those listed earlier in this section and in Chapter 2.2).

There are two relations between the resource owner and resource users that I explore in this dissertation, as depicted in Figure 4.2. These relations have implications on cost metrics and the prevention of resource abuse that are described in Chapter 4.2. Both are common and have historical precedent. In the first, the resource owner and resource user are the same person, organization, entity, called a *cost-center,* and computing time is not directly charged; such as when a laboratory buys a resource and its members cooperate in using the resources toward some larger goal. In the second, computing time is sold to another party; the resource owner is an outsourcer, IT manager, or *profit-center* with no interest in task output.

Profit-centers are known by many names, such as third-party compute outsourcing, resource hosting, IT (information technology) resource provider, off-site resource provider, resource virtualization, on-demand computing, etc. IBM's service is called Strategic Outsourcing [IBM, 2004]. Weidenhammer Systems Corporation, an IT firm, provides what they call Outsourcing/Hosting [Weidenhammer, 2004]. EDS provides Application Selective Outsourcing [EDS, 2004]. Hewlett-Packard's service was the Utility Data Center [Hewlett-Packard, 2004] (decommissioned in the year 2004).

[2] I helped develop the research clustering software called GLUnix [Ghormley et al., 1998], one component of the UC Berkeley Network of Workstations (NOW) Project [Anderson et al., 1995].

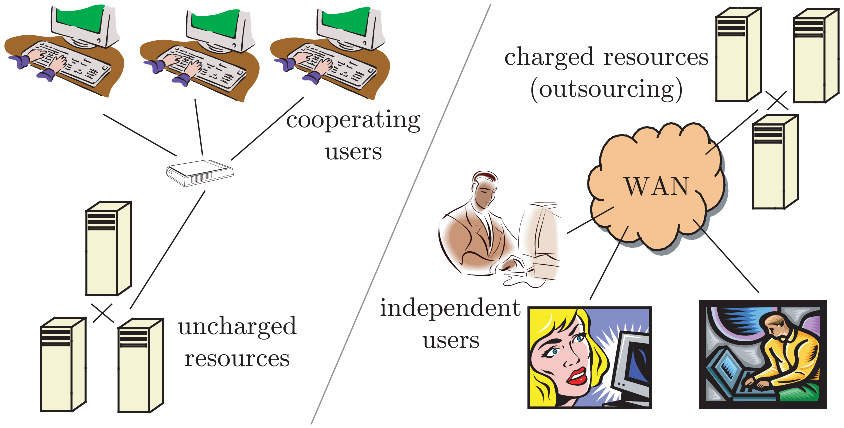


Figure 4.2: Two relations between the resource owner and resource users. On the left is depicted uncharged resource usage, occurring when users are cooperating toward a shared goal or taking part in resource ownership; such resources are often accessed through a local area network. On the right is depicted charged resource usage, when a number of independent users buy resource time from a usually geographically distant set of resources over a wide area network.

4.2 Cost model

In this section I describe resource pricing in the context of existing non-speculative scheduling environments. When describing my batchactive environment, I introduce a new pricing mechanism more appropriate for speculative tasks (Chapter 5.1).

The resource pricing mechanism of these services vary based on limitations of accounting, associated application set-up costs, whether resources are charged on a per-unit or time basis, whether more or less task throughput is a consideration, etc. Further, the cost to use some amount of a resource can be constant or variable. Sometimes an intermediate, abstract currency is used: resource usage might be equated to *service units,* which themselves have a monetary price.[3]

If the same self-interested entity (individual, laboratory, institution) both owns and uses the resource, then the resource is not directly charged; it is a cost-center. An example is when a laboratory buys a resource and its members cooperate in using the resources toward some goal. The resource

[3] At Georgia Tech, service units are colloquially referred to as bananas; i.e., x bananas for some mid-sized Unix server usage, y where $y > x$ bananas for some Cray supercomputer usage, z bananas for some disk space usage, etc.

is payed by some entity, but the users of the resource do not directly pay for the resources that they use: resource usage is unaccounted and there are no user quotas on resource use. If computing time is policed or controlled, it is done so by means other than charging for its use (as discussed further in Chapter 5.8). 'Free'[4] resource usage has historical precedent: e.g., the university settings of clusters and the ownership of a cluster by a computer graphics company that employs graphics artists who use these clusters. To break even, a cost-center would set the price per resource unit to be its operational cost per time[5] (such as a month) divided by the number of expected resource units it will sell in that time. Individual decisions by users to submit tasks may not be directly charged, but eventually money is moved within an organization to cover the costs of the communal resource.

Assuming that the cost to provide computational resources is largely elastic (meaning that it costs the resource provider the same to provide no resources or full utilization of its resource over some time), if the resource provider is a profit-center and wishes to maximize profit (which is revenue minus cost), it should set the price to that which maximizes the multiplicative product of price and quantity demanded from a demand schedule. More detailed economic considerations in setting price or in finding demand curves is beyond my scope.

Variable pricing, by being more dynamic, might lead to a more efficient pricing mechanism. Hewlett-Packard introduced the 'computon,' a service unit whose price changes with resource supply and demand, often loosely mentioned in analogy to the pricing mechanism used by electrical utilities [Hoffman, 2003]. This mechanism applies the economic theory of congestion or shadow pricing [MacKie-Mason and Varian, 1995] which measures the extent to which a task's resource usage takes that resource away from other tasks.

On a supercomputer, resource time is typically applied for as one of the rewards of a grant [Lemieux, 2003]. Although the user may not pay for resource usage, the organization (often a government) offering the grant did pay for it when it bought the right to use some resource time from the resource owner. Because governments often provide grants, fund the building of supercomputer resources, and pay researcher salaries, and because

[4] A free good is one in which the supply is at least equal to the demand at zero price; viz., it is not scarce [Baumol and Blinder, 1994]. An unaccounted resource is not free in this sense because there is a cost to waiting for the resource to become available if it is under contention, e.g.

[5] Operational costs may include paying back a portion of the initial hardware investment, electricity, hardware / software maintenance, machine room rent, and so on.

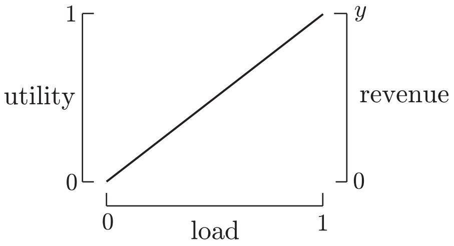


Figure 4.3: How load affects server utility and revenue under the pricing mechanism typical under non-speculative scheduling of charging a constant amount for resource usage. (Figure 5.2 is the analogous sketch for speculative scheduling.) Load (or efficiency), the fraction of time a resource is busy, varies between 0 and 1. Server utility or revenue is tied to the amount of resources a server charges. As load increases, utility increases equally. Revenue can be calculated directly from load given a cost per unit resource. Shown at maximum utility (1) is maximum server revenue (y).

the government has interest in the scientific output from these resources, supercomputer centers can often be considered cost-centers.

This thesis studies the implications that different schedulers have on resource costs, both from the user and resource provider's perspective. In wake of the listed pricing mechanisms, I assume that a resource provider's utility is linearly proportional to the amount of resources used during some time period. That is, a resource provider is best off if it was busy executing tasks 100% of the time, and does only half as well if it was busy 50% of the time. Thus, a resource provider desires high load (where load is the fraction of busy time, also called utilization or efficiency). If resources go unused, then those resources did not produce revenue. This linear correlation between resource usage and the utility of a resource provider is most natural under constant pricing, which is the most common pricing mechanism. This relationship between load and server utility is depicted in Figure 4.3.

My simulation results on how scheduling affects user costs and resource provider utility (Chapter 6.2) tracks the individual and total resource time consumed across users over some time. With this data of resource usage under different schedulers, one can apply additional information (such as price per resource second, the resource provider's operating overheads for idle v. busy resources, etc.) to obtain cost, revenue, and profit figures for users and the resource provider tailored for any particular situation. Because it is more important for this thesis to study the relative difference among scheduling approaches, specific values are not discussed.

4.3 Definitions and metrics

First I define the entities in non-speculative scheduling and how they interact. Then I introduce the scheduling metrics needed in Chapter 4.4 to define scheduling goals. These definitions and metrics are refined in my batchactive scheduling environment in Chapter 5.2.

There are three sets of entities called *servers, users,* and *tasks.* There are a finite number of elements in each set. Each server can run computational work in the form of a single task serially. (An actual machine with more than one processor is logically multiple servers.) The set of servers is called the *computing system* or *cluster resources.* A user is a self-interested person or agent that *arrives,* interacts with the computing system, and *departs,* in this order. There can be zero, one, or more users interacting with the system depending on when they arrive and depart.

In this section I refer to the set of tasks as A. A task (also called a job) $a \in A$ represents some work associated with a specific user that can use one server. A task can be requested, canceled, or executed, exclusively.

An arrived user can request the output of one or more tasks. (A departed user cannot.) For simplicity, I say interchangeably that a user requests a task or that a user requests a task's output. Let t_a^r denote when a was requested. The set of *requested tasks* up to time t is denoted by $A^r(t)$.

A user can cancel tasks that he or she had previously requested, but that not have already been executed or canceled. Also, the system cancels any requested tasks by departed users. A canceled task remains forever canceled. The set of *canceled tasks* up to time t is denoted by $A^c(t)$. A canceled task becomes no longer requested. That is, $a \in A^c(t) \implies a \notin A^r(t)$.

Each task $a \in A$ has a corresponding *service time* (also called size or resource requirement) S_a which is constant and a *resource usage* $r_a(t)$ at time t, where $S_a > 0$ and $0 \leq r_a(t) \leq S$. Service time and resource usage are reals and the unit for both is time.

Requested and non-running tasks are candidates for the *scheduler* to choose. A task runs at time t if and only if the scheduler decided so. The set of *running tasks* is denoted by

$$A^\star(t) \overset{\text{def}}{=} \{a \in A^r(t) \mid a \text{ runs at time } t\}\,.$$

When there is only one server in the computing system, $|A^\star(t)| \leq 1$.

The task's resource usage is 0 at t_a^r and increases by the amount of time that it runs. That is, if a task ran for δ amount of time, its resource usage

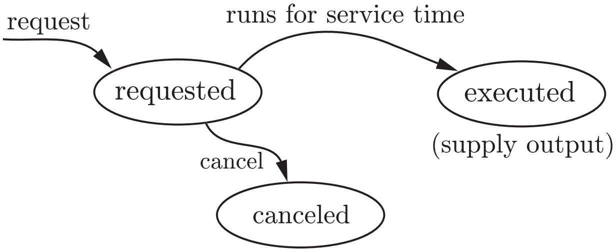


Figure 4.4: Task state transitions with a non-speculative scheduler (compare to Figure 5.5). After the requested task's resource usage equals its service time, it becomes executed, and the requesting user receives its output. A requested task can be canceled before it executes.

increases by δ. Formally,

$$\begin{gathered}\text{if } a \in A^r(t_a^r), \text{ then } r_a(t_a^r) = 0,\\ \text{if } a \in A^\star(t), \text{ then } r_a'(t) = 1,\\ \text{if } a \notin A^\star(t), \text{ then } r_a'(t) = 0.\end{gathered}$$

Let t_a^e, where $t_a^e > t_a^r$, denote the time at which r_a grows to equal S_a. (Of course, a task's service time may be unknown until a task completes [Turing, 1936].) The task is considered from this time on to be executed. The set of *executed tasks* up to time t is denoted by

$$A^e(t) \overset{\text{def}}{=} \{a \in A \mid r_a(t) = S_a\}\,.$$

When a task becomes executed, the computing system provides the task's output to the requesting user, and the scheduler removes the task from the set of requested tasks. That is, if $a \in A^e(t)$, then $a \notin A^r(t)$.

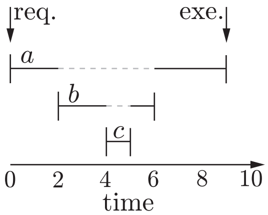
See Figure 4.4 for a pictorial representation of the states in which a task can reside in non-speculative scheduling.

Batchactive scheduling is not completely described by these definitions which do not represent the disclosure of speculative tasks a user might request in the future.

Now I describe scheduling metrics, which are used to evaluate how well the scheduler achieves the scheduling goals elaborated upon in Chapter 4.4.

Response time (flow time, sojourn time, time-in-system) is a main scheduling metric. A task a requested at time t_a^r and executed (completed) at time t_a^e has a corresponding *response time* denoted by

$$T_a^{\text{resp}} \overset{\text{def}}{=} t_a^e - t_a^r.$$



	a	b	c
service time	5	3	1
response time	9	4	1
slowdown	1.8	1.3	1

Figure 4.5: How when a task is requested and executed, along with a task's service time, determines its response time and slowdown in the context of non-speculative scheduling. (Compare to Figure 5.6.) Three single-threaded tasks are shown in on a single processor model. In this particular scheduling policy, shorter tasks preempt (indicated by a dotted line) larger tasks.

An executed task with service time S_a also has a corresponding *slowdown* (also called stretch) denoted by

$$T_a^{\text{slow}} \overset{\text{def}}{=} \frac{T_a^{\text{resp}}}{S_a}.$$

Response time and slowdown are depicted in Figure 4.5.

Across all tasks that have executed by some time, *mean response time* and *mean slowdown* are important scheduling metrics.[6] Another metric is the *variance of response time.*[7]

In the context of speculative scheduling, in which users are able to submit possibly tentative computational plans in advance and only need task outputs later due to think time, a task may execute *before* requested and these metrics become insufficient. I introduce *mean visible response time* and *mean visible slowdown* in Chapter 5.2 as a refinement to mean response time and mean slowdown.

Task *throughput* is the number of tasks completed during some time period. I track a variant of this quantity that ignores speculative tasks that were eventually found to not be needed (Chapter 6.2) to confirm that improving other metrics (such as response time-based metrics introduced in Chapter 5.2) does not pathologically cause fewer needed tasks to complete.

[6] Recall that the mean of a sample set, in this case where each 'event' is equally likely, is simply the arithmetic average of the set: $\frac{1}{N}\sum_{i=1}^{N} x_i$.

[7] Recall that the variance of a sample set is the sum-of-squared differences of elements from this set and the mean of the set, over the number of elements in the set: $\frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2$. This definition is a 'biased' estimator of variance, because both the mean and variance are estimated from the samples simultaneously. To obtain an unbiased estimator of variance, the following $N-1$ correction is made, as explained by Weisstein [2004g]: $\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \bar{x})^2$.

Consider the total resources used by a user. The variance of this value across users is called the *variance of user resource usage.* This is another studied metric that reflects the extent to which users use different amounts of resources. The lower this variance, the closer users are to using the same amounts of resources over long time periods.

Although a server can only run one task at a time in this model, the following metric shows the extent to which requested tasks (i.e., each candidate for the scheduler to choose) received different amounts of a server's resource at small time intervals. Consider the fraction of a server's resource used by a task at a specific time t. Each requested task $a \in A^r(t)$ has a corresponding instantaneous slowdown which is the inverse of this fraction, i.e.,

$$\begin{aligned} &\lim_{\delta \to 0} \frac{1}{\text{fraction of time } a \text{ ran during } [t, t+\delta]} \\ &= \lim_{\delta \to 0} \frac{\delta}{r_a(t+\delta) - r_a(t)}. \end{aligned}$$

The *variance of instantaneous slowdown* for requested tasks shows the extent to which resources have been provisioned equally among competing tasks. That is, the lower this variance, the closer tasks are to equal-share (a type of fair-share) over short durations.

The *ratio of maximum slowdown to mean slowdown* reflects the extent to which 'starvation' exists; i.e., whether a few tasks receive considerably less service than others. This definition is my own. Some look at maximum slowdown across all tasks [Bender et al., 1998], others bin tasks based on service time and look at the maximum slowdown for all tasks in a particular bin [Bansal and Harchol-Balter, 2001].

I measure the billed resources wasted on tasks that were later known to not be needed (i.e., requested tasks that were canceled after they had consumed some resources). Users who believe that possibly minimizing response time is worth the cost will submit work speculatively to non-speculative schedulers. I track such users' *scaled billed resources,* the ratio of the billed resources to the needed resources. For example, if a scheduler charged a user for ten seconds of resource time but the user only needed five seconds of resource time, then the scaled billed resources is 2. Across all users, I measure the mean scaled billed resources. Anytime a user using a non-speculative scheduler that charges for all resource use submits a task speculatively that he or she did not eventually need, the mean scaled billed resources will be above 1.

metric	description
mean response time	average time between request and execution
mean slowdown	average response time scaled by task size
task throughput	number of completed tasks
variance of response time	how response times differ
variance of user resource usage	how per-user resource usage differs
variance of instantaneous slowdown	a measure of equal-share
maximum over mean slowdown	a measure of starvation
mean scaled billed resources	average per-user billed over needed resources
load	fraction of server busy time
decision count	number of scheduling decisions

Table 4.1: Non-speculative scheduling metrics (compare to Table 5.1 for speculative scheduling metrics).

Again, for a user who never submits a task before needing its result, no resources are wasted, and thus the user is not billed for any wasted resources. But when a user who speculates with a non-speculative scheduler does not need the outputs from all submitted tasks, the user will be charged for the resources used for unneeded tasks. From the user's perspective, he or she was charged needlessly for wasted resources.

For each server in the cluster, I track its *load* (also known as device utilization), the fraction of time that the server was busy (running some task) during some time period. Thus, load on this single processor is a real between 0 and 1.[8] Recall that load is directly proportional to the resource provider's utility or revenue (Chapter 4.2).

The number of scheduling decisions made in some time period is its *decision count.* It is important that a policy not cause substantially more decisions to be made than another policy, because each decision leads to two types of overhead: time overhead in making a scheduling decision and potentially overhead in switching context between one task and another if the policy determines that another task should run.

The metrics introduced in this section that are used in other parts of this dissertation are summarized in Table 4.1.

[8] In non-formal contexts, task queue length is sometimes called load (such as reported by the `uptime` command on Unix). However, I use the queuing-theoretic definition of load in which a load of 1 means that the server is totally busy, that there is no idle time; and thus load cannot exceed 1.

4.4 Scheduling goals

Users and resource providers are self-interested entities that have scheduling goals. The scheduling goals among them differ and sometimes conflict. In fact, some of the user goals listed below conflict among themselves.

In this section I describe user and resource provider scheduling goals. When the resource user and owner are the same entity, goals related to billed resources do not apply (such as a resource provider's desire for high server load). The scheduling metrics that I refer to in defining these goals were formalized in Chapter 4.3.

4.4.1 User goals

In the abstract, users wish to maximize utility: the difference between the value and cost of a commodity to the user. The commodity is task output. The user wishes to maximize the most outputs in the best order during some time period. However, the value of task outputs is hard to define. All a scheduler can do is return task outputs faster or slower or in some new order. In general, a scheduler cannot know the value of outputs to a user to make ideal scheduling decisions, or even know the order in which the user would prefer to receive outputs. Users themselves have a hard time estimating value; moreover, the value of any particular task may not be known for some time. In the context of non-speculative scheduling, a user requests a number of tasks in some order; *the scheduler assumes that each task is equally important to the user and that the user would prefer outputs to return in the requested order.*

What is known to contribute to the value of a task's output and what is universally accepted as an important scheduling metric [Conway et al., 1967, ch. 8] [Endo et al., 1996] (and what also, fortunately, is easily measurable) is the time between output being ready for the user and the user requesting the output — i.e., task response time. Because knowing a task's value accurately for all situations is impossible, in this discussion of user goals, a scheduling goal is to minimize the response time of a task. Across all tasks in some time period, the primary scheduling goal is to *minimize mean response time.* Note that maximizing throughput is not an acceptable substitute for minimizing mean response time from the user's perspective. The philippic of Endo et al. [1996] and my discussion of policies that affect load in Chapter 4.5.4 show that throughput does not directly correlate with mean response time.

Recent scheduling work [Feitelson and Jette, 1997; Bender et al., 1998; Harchol-Balter et al., 2002] argues that slowdown is more important to minimize than response time for the following two reasons: (1) slowdown expresses the notion that users are willing to wait longer for larger amounts of work, and (2) mean slowdown better reflects the performance of most tasks instead of just a few large tasks for the common case [Harchol-Balter and Downey, 1997] in which the distribution of service time is heavy-tailed (Chapter 6.1.3). Thus, another goal is to *minimize mean slowdown.* These first goals conflict: it is not true that the scheduling order which minimizes mean response time also minimizes mean slowdown [Harchol-Balter, 2003b].

In the context of speculative scheduling, in which a task may execute *before* requested, these goals are insufficient. I advocate *minimizing mean visible response time* and *minimizing mean visible slowdown* in Chapter 5.3.1 as a refinement to these goals.

Utility also concerns cost. When billed, the cost for task output is the price (the cost expressed in money) the user has to pay for the resources used to produce the output. The user wishes to minimize what he or she pays over some time period when resources cost. According to the pricing mechanism described in Chapter 4.2, resource cost is proportional to resource usage. It is not a scheduling problem to reduce user costs (to minimize mean scaled billed resources): the scheduler has no control over what tasks a user requests and the service times of these tasks. The user had predicted for him- or herself that the value of the task's output would be more than the price. (In the parlance of rational users, this would be that the user had predicted that his expected utility would be positive in requesting the task.) While minimizing mean scaled billed resources is not a goal that a scheduling policy can seek, the metric mean scaled billed resources is reported (Chapter 6.2) to compare the effect of the differing pricing mechanisms in the non-speculative (Chapter 4.2) and speculative (Chapter 5.1) scheduling environments.

In environments in which resource usage is not directly charged, the following goal, while not related to utility, pressures a user to not use substantially more resources than other users (in other words, to discourage resource abuse): strive to equalize the resource usage of all users at every scheduling decision. This goal conflicts with minimizing mean response time and minimizing mean slowdown. Further, this goal is trivially achieved by never running any task. Thus, it is better phrased as *minimizing the variance of user resource usage while remaining work-conserving.*[9] Because there are situations in which this goal is more important than time-based goals, I also study speculative and non-speculative schedulers that attempt to achieve it.

I now introduce, for completeness, secondary goals. Because they are less

[9] Work-conserving: a server will never be idle if there a task is ready to run.

important, no scheduling policies are designed to meet them in this thesis.

Users dislike response time variance, especially on small time scales. It has been shown that users would prefer some task outputs to be delayed so that response time variance is smaller [Shneiderman, 1997, ch. 10.5]. This goal, which conflicts with minimizing mean response time and mean slowdown is to *minimize the variance of response time.* Because minimizing response time variance is less important at the larger time scales of my target application domain — the scenarios in Chapter 2.2 include tasks that typically take minutes and longer, rather than seconds and shorter — I do not discuss this goal further. The variance of response time, in a different form, is measured in the results of this thesis (Chapter 6.2) to confirm that it is not made worse when scheduling with the novel speculative schedulers introduced in the following chapter (Chapter 5).

Many hold that all tasks that are candidates for running should have the same instantaneous slowdown at every point in time; that the scheduler should provide the same fraction of the resource to all candidates regardless of their resource usages, service times, etc. This goal is not inherently worthwhile and may be popular because it is easy to understand and achieve, requiring no information besides the list of candidates. Perfectly *minimizing the variance of instantaneous slowdown* is a definition of equal-share (a type of fair-share) scheduling. While this goal has been traditionally studied, I do not address it in this thesis.

Finally, a common user scheduling goal is for no task to 'starve,' i.e., tasks should experience slowdown within some small factor of the mean slowdown. Thus, a scheduler should *bound the ratio of the maximum slowdown to the mean slowdown.* Across a variety of loads and among different schedulers, starvation has not been a problem in my experiments (Chapter 6.2), thus it is not explicitly focused on in this thesis. Further, among the fundamental non-speculative schedulers discussed below (Chapter 4.5), which are also used as building blocks for batchactive scheduling (Chapter 5.5), only one (SRPT) has the potential for starvation and recent analysis shows that any concern for starvation by using this policy over another is unfounded [Bansal and Harchol-Balter, 2001].

4.4.2 Resource provider's goals

From the resource provider's perspective, according to the pricing mechanism described in Chapter 4.2, a scheduler should *maximize load*, the amount of time that the resources are busy. Every cycle of computation consumed by a user is billable, whether directly for a profit-center, or indirectly for a

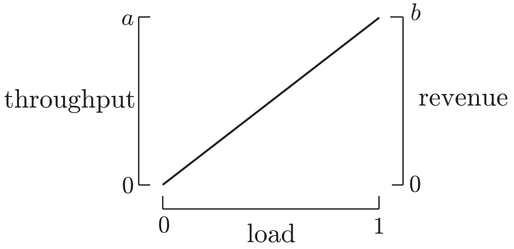


Figure 4.6: How load affects throughput and revenue under non-speculative scheduling. (Compare to Figure 5.7.) As load varies, throughput and revenue vary proportionally. Throughput is load scaled by mean task service time (Utilization Law) and revenue is load scaled by mean task service time and the cost per unit resource. Shown at maximum load (1) is maximum throughput (a) and maximum server revenue (b). This sketch does not depict the effect of task cancelation: tasks which may have taken some load before being canceled should not increase throughput.

cost-center. When a resource is idle, the provider loses revenue that would have been used to meet the resource owner's revenue goal, whether it was profit maximizing or non-profit (trying to break even). (Resource provider utility can also improve by lowering costs. It is beyond my scope to address how a resource provider could reduce its costs in producing outputs, such as by using equipment that is less costly to maintain.)

Sometimes it is stated that the resource provider's goal is instead to maximize task throughput. These goals are interchangeable for the non-speculative pricing mechanism (Chapter 4.2). From the Utilization Law (Chapter 4.5.4), load is the multiplicative product of throughput and mean task service time [Harchol-Balter, 2003b], i.e., load is throughput normalized by mean service time. It is more natural in comparing server revenue among schedulers, user behavior, and task characteristics to consider load, which varies between 0 and 1, than to compare throughput, which can vary widely, as shown in Figure 4.6. In the next chapter (Chapter 5.2), throughput is refined to a 'visible throughput' of the tasks actually needed by users (i.e., speculative tasks eventually known to not be needed are omitted), and load is refined to a 'requested load' of tasks actually billed under the batchactive pricing mechanism (Chapter 5.1).

Maximizing load consists of (1) encouraging users to use the resource provider's service over competing services and (2) encouraging users to submit as much work as quickly as possible.

Encouraging users to use the service is a matter of meeting user goals to ensure return customers and increase customer base. It is in the resource

with respect to	goal
user	minimize mean response time
user	minimize mean slowdown
user	minimize the variance of user resource usage
resource provider	maximize load

Table 4.2: Non-speculative scheduling goals. (Compare to the speculative scheduling goals of Table 5.2.)

provider's interest to meet user goals. To increase the number of users, a computing system needs to promote its services and provide value above competing services. Measuring the success of a resource provider in doing so is outside the scope of this thesis. However, as discussed along with speculative scheduling goals, (Chapter 5.3), the intended and measured benefits to user scheduling goals obtained through supporting speculative tasks as first-class entities provides incentive for a user to select a batchactive system over a traditional system.

The speed at which users submit work is a function of user think time, which a scheduling policy cannot control, and a function of how fast task outputs are supplied to the user, which *is* determined by scheduling policy. Both of these considerations are elaborated upon in Chapter 4.5.4 and measured (by a parameter study of user behavior and by selecting among different schedulers) in the results of this thesis (Chapter 6.2).

4.4.3 Summary of scheduling goals

I discussed user goals and resource provider goals for non-speculative scheduling. The goals were built on the scheduling metrics listed in Table 4.1. For both users and a resource provider, I began in the abstract: users wish to maximize the most outputs in the best order and minimize what they pay for such outputs over some time period. A resource provider wishes to maximize its revenue. These abstract goals, along with the pricing mechanism in Chapter 4.2 lead to the practical goals listed in Table 4.2, as I have argued and as are generally accepted by scheduling literature.

Minimizing mean response time and minimizing slowdown sometimes conflict because the algorithm to achieve one does not achieve the other. However, the optimal algorithm for minimizing mean response time *among the policies discussed next* (Chapter 4.5) does best at minimizing mean slowdown (Chapters 4.5.1 and 4.5.2). Minimizing the variance of user resource usage is employed where users should be cooperating with shared resources whose use is not directly charged to the user but where such cooperation cannot be assumed. This goal works to prevent users from dominating resources. Maximizing load maximizes resource provider revenue.

I do not address the following secondary goals in this thesis: minimizing the variance of response time, minimizing the variance of instantaneous slowdown, and bounding the extent of task starvation.

4.5 Policies in theory

A process scheduler switches among competing demands of a cluster's processor resources. Here I describe how well several fundamental policies meet the scheduling goals listed in Table 4.2. These are *online* policies, meaning that the existence of a task is revealed to the policy only when it arrives, and thus the policies make decisions without knowing what tasks will arrive in the future.[10]

The number of studied online scheduling policies is large even when omitting types of scheduling (like real-time) that are outside my scope (Chapter 3). What I present are typical deployed building blocks. Priority-based or classed scheduling is covered in the context of batchactive scheduling (Chapter 5.6.2).

After describing these policies in theory, the following section (Chapter 4.6) shows how they are adapted for use by resource providers.

Consider the following policies:

– *First-come-first-serve* (FCFS).

  The earliest requested task runs until completed.

– *Processor-sharing* (PS).

  All requested tasks run simultaneously.

– *Foreground-background* (FB).

  The requested task with the lowest resource usage (least attained service) runs. If several such tasks have the same lowest resource usage, those tasks run simultaneously.

[10] *Offline* policies, which know the entire task arrival sequence in advance, do not apply to the dynamic, real-world application domains consisting of users that may arrive and submit work at any time.

– *Shortest-remaining-processing-time* (SRPT).

The requested task with the least remaining work runs.[11]

Other policies, e.g., last-come-first-serve and shortest-job-first, are omitted because their behaviors are either not significantly different or are worse than the behaviors of the listed policies for the goals under consideration.

All but SRPT are *non-clairvoyant;* they do not make use of task size. All but FCFS are preemptive; they are not restricted to running tasks to completion. Because SRPT is biased toward tasks with less work remaining and because it is preemptive, there is the possibility that large tasks will starve. However, this fear is unfounded [Bansal and Harchol-Balter, 2001].

The conditions (such as service time distribution) under which one policy is better than another for a particular goal is sometimes an open scheduling question. Definitive statements below assume task arrivals that follow a Poisson process,[12] a general service time distribution, and a single server, unless otherwise noted.

4.5.1 Concerning mean response time

It has been proven that SRPT optimally minimizes mean response time [Conway et al., 1967, ch. 8] [Schrage, 1968]. Since response time accrues as soon as a task is requested, completing a task sooner leads to lower response time for that task. Doing so overall, by choosing the task with the least remaining work, leads to lower mean response time compared to other policies. This intuition is also helpful in understanding the behavior of the other policies.

The service time distribution's failure rate [Weisstein, 2004b] (also called hazard rate) determines the best policy for minimizing mean response time when SRPT cannot be employed. The failure rate of a distribution is $\frac{P(x)}{1-D(x)}$, where $P(x)$ is its probability density function and $D(x)$ is its cumulative distribution function. In this context, x is the amount of resources used by a task at some point in time. If the failure rate is increasing in x, then the distribution is said to have an *increasing failure rate* (IFR); if decreasing in x, then a *decreasing failure rate* (DFR); and if independent of x, then a constant failure rate (CFR and also called the *memoryless* property). Failure rates can be understood by analogy. For IFR, the longer a car is driven, the

[11] In Chapter 4.7 I discuss how task size, which is necessary for knowing the amount of remaining work, can be predicted.

[12] A Poisson process [Weisstein, 2004e] with rate λ is a sequence of events such that the times between events are independently selected from an exponential distribution (Chapter 6.1.3) with rate λ.

less time it is expected to last. For DFR, the longer a light bulb is on, the more time it is expected to last. Finally, for CFR, a radioactive atom has the same probability of decay at any time, regardless of how long it has existed.

When the service time distribution has an increasing failure rate, such as is found in a uniform distribution, FCFS was proven to achieve a lower mean response time than PS, and PS was proven to achieve a lower mean response time than FB. The more resources a task has consumed, the sooner it will end, and thus it should keep running, which is what FCFS does better than PS, and what PS does better than FB. When the service time distribution has a decreasing failure rate, such as is found in a Pareto distribution, FB was proven to achieve a lower mean response time than PS, and PS was proven to achieve a lower mean response time than FCFS. The more resources a task has consumed, the later it will end, and thus it should be avoided, which is what FB does better than PS, and what PS does better than FCFS. Finally, when the service time distribution has a constant failure rate, a property held only by the exponential distribution, FCFS, PS, and FB were proven to perform the same with respect to mean response time. The resources a task has consumed is irrelevant to when it will end. [Harchol-Balter, 2003b; Wierman et al., 2004; Wierman, 2004][13]

It has been shown that task service time can be often modeled well by a Pareto distribution [Harchol-Balter and Downey, 1997]. Since Pareto distributions have a decreasing failure rate, FB is the best choice among FCFS, PS, and FCFS.

If the service time distribution's failure rate is not known but its variance and mean are known, the following result determines whether PS or FCFS is better. PS, which optimally minimizes the variance of instantaneous slowdown at every point in time, was proven to achieve lower mean response time than FCFS if and only if the squared coefficient of variation [Weisstein, 2004h] of task service time is greater than 1; i.e., if and only if $\mathrm{var}(S)/\bar{S}^2 > 1$, where $\mathrm{var}(S)$ is the variance of task service time and $\bar{S}$ is the mean task service time.[14] [Harchol-Balter et al., 1997]

4.5.2 Concerning mean slowdown

Optimally minimizing mean slowdown is an open problem (and active research area). Among FCFS, PS, FB, and SRPT from Chapter 4.5, SRPT performs best [Bender et al., 2002; Bansal and Dhamdhere, 2003]. There are

[13] The mentioned distributions are detailed in Chapter 6.1.3.

[14] The squared coefficients of variation of the uniform, exponential, and Pareto distributions are less than 1, 1, and greater than 1, respectively. [Harchol-Balter, 2003b].

better algorithms for minimizing mean slowdown, but how doing so affects mean response time is not well understood and thus not discussed further.[15]

4.5.3 Concerning the variance of user resource usage

A modification to FB leads to a policy that optimally minimizes the variance of user resource usage while remaining work-conserving. Recall that FB runs the task with the lowest resource usage. Instead of looking at a task's resource usage, a user-based FB policy, which I call user-FB, looks at the total amount of resources consumed by a user. Thus, when a scheduling decision is to be made, user-FB selects the task from the user that has used the fewest resources. If several users have the lowest resource usage and have queued tasks, then the their tasks are run simultaneously (i.e., using PS, although implementations usually approximate this by only making decisions on events such as a task being submitted or finishing). If a user has more than one task queued, those tasks are taken in FCFS order from that user under the assumption that the order of submission reflects the order of need. This assumption fits the sequential tasks application type (Chapter 2.2.2).

4.5.4 Concerning load

Load on a single server is the multiplicative product of task throughput and mean task service time according to the Utilization Law and sketched in Figure 4.6 [Harchol-Balter, 2003b]. I assume that the performance of a scheduling policy cannot affect mean task service time (a function of how big the tasks that users submit are, the speed of the server, and whether a policy is work-conserving), so changing mean service time to affect load is not discussed in this section.

The extent to which a scheduling policy can influence throughput (and thus load) depends upon the manner in which people use the computing system. Queuing theory models systems as either open, in which task arrivals are independent of scheduling, or closed, in which a constant number of users submit one task at a time, wait for task output, think about the output for some *think time,* and repeat. In my target application domain (Chapter 2.1), how people use computing systems combines aspects of an open system (users arriving and departing) and a closed system (users needing and thinking about the output of one task before the next task).

[15] A scheduling theorist confirms the lack of analysis in the literature for how FCFS, PS, and FB compare with one another with respect to minimizing mean slowdown, but intuitively believes that the same conclusions in Chapter 4.5.1 concerning their relative merit for response time also hold for slowdown [Wierman, 2004].

For an open system, a higher mean task arrival rate (which can be realized with more users) means higher throughput and thus higher load, so long as the system does not go into overload (load > 1). Task throughput is, in fact, equal to the mean arrival rate for open systems [Harchol-Balter, 2003b] so long as the mean arrival rate is not greater than the mean service rate, i.e., the rate that tasks complete.[16] Observe that any work-conserving policy exposed to the same task arrival sequence (viz., when tasks arrive and the sizes of those tasks) on the same speed server leads to the same load [Harchol-Balter, 2003b]. Because arrivals occur independent of when tasks complete, the 'work in system' is the same. Since the scheduling policies listed above are work-conserving, the load is the same for them all when exposed to the same arrival sequence, assuming an open system.

For a closed system, higher throughput can also — up to the point mentioned below — be obtained with more users. How to encourage more users to use one resource provider over another (irrespective of whether the system is open or closed) was discussed in Chapter 4.4.2.

For a closed system, combining the Utilization Law and Little's Law shows that two additional considerations beyond the number of users affect load (and throughput). These laws show that load is inversely proportional to the sum of mean response time and mean user think time [Harchol-Balter, 2003b]. A scheduling policy cannot affect the amount of time users spend thinking of task outputs.[17] However, mean response time *is* a property of scheduling effectiveness. The intuition of how mean response time can affect load is that, in a given time period, users who receive task outputs faster submit more work. Thus, a policy which reduces mean response time (or increases the number of users, as stated above) also increases load (and server utility or revenue).[18] Here, it is not true that every work-conserving

[16] The Utilization Law states that $U = XE[S]$, where U is load, X is throughput, and $E[S]$ is the expected (mean) task service time. In an open system, $\lambda = X$ (tasks in equals tasks out), where λ is the task arrival rate, so long as $\lambda \leq \mu$, where μ is the mean service rate equivalent to $1/E[S]$. Thus, $U = \lambda E[S]$. Assuming mean task service time cannot change (i.e., the processor speed cannot change, the tasks do not change size, and the scheduling policy is work-conserving), load can be increased with a higher task arrival rate (such as by adding more users). As λ approaches μ, where $\mu = 1/E[S]$, load approaches a maximum of 1.

[17] As a less significant effect, user think time might be correlated with task response time. If mean response time was low for a long period of time, the user might become fatigued and exhibit longer think times. This effect is not considered further in this thesis.

[18] Again, the Utilization Law states that $U = XE[S]$, where U is load, X is throughput, and $E[S]$ is the expected (mean) task service time. Little's Law for a closed system states that $E[R] = N/X - E[Z]$, where $E[R]$ is the expected (mean) response time, N is the number of users, and $E[Z]$ is the expected (mean) user think time. Combining both laws

policy is equally profitable. Fortunately, minimizing mean response time does not conflict with the user goals; in fact, it is a major user goal. Above (Chapter 4.5.1) I discussed how different policies compare toward minimizing mean response time.

In sum, if users behave more like an open system (with user arrivals and departures independent of when tasks are serviced), then to increase load the task arrival rate must increase, which is practically realized by adding more users to the system. Adding users to increase throughput would affect load in an open system to the same degree across scheduling policies, although other metrics, like mean response time, would likely change to different degrees [Harchol-Balter, 2003b]. If users behave more like a closed system, then in addition to adding more users to increase load, a scheduling policy which does well at minimizing response time should be employed.

There are other considerations outside the scope of this thesis when tasks need more than one processor at once ('space-shared' applications). For example, one approach to improving load is to favor long-running tasks that require a large number of nodes (to the detriment of response time) because such tasks would otherwise have difficulty getting the number of nodes necessary to run [Lemieux, 2003]. More information can be found within the scope chapter (Chapter 3.2).

4.5.5 Summary of policies in theory

The following approach does well for minimizing mean response time and maximizing load and does not do badly for minimizing mean slowdown: If task size is known, apply SRPT. If task size is unknown, but the resource provider will attempt to predict it, use SRPT along with one of the procedures outlined in Chapter 4.7. If predictions will not be employed, then a policy that does not require knowing task size must be used. Because task size distribution has the DFR (decreasing failure rate) property [Harchol-Balter and Downey, 1997], the next best policy among those considered is FB.

If the resource provider needs to discourage users from abusing resources (when a small subset of users use substantially more resources than others) then user-FB should be employed to minimize the variance of user resource usage while remaining work-conserving. Here, the other goals (minimizing response time and slowdown and maximizing load) become secondary. The theoretic effect of user-FB on these now secondary goals is unknown to me but is measured in my results (Chapter 6.2.7).

leads to $U = \frac{NE[S]}{E[R]+E[Z]}$. Assuming task service time and user think time cannot change, load can be improved with more users and with scheduling that lowers $E[R]$.

4.6 Scheduling in practice

Applying the fundamental scheduling policies described above (Chapter 4.5) to a cluster or supercomputer center involves additional considerations. Some resistance to direct application is due to prejudice (i.e., SRPT starves large tasks, a belief that has been rebutted [Bansal and Harchol-Balter, 2001]) or lack of information (i.e., SRPT needs to know task sizes). But the resistance is more fundamental: There are additional goals beyond those derived in Chapter 4.4 from abstract notions of utility; there are realities in balancing a pricing mechanism, application development (debug) cycles, tasks with special hardware needs, users with special priority needs, and imperfect accounting.

Scheduling in practice is a hybrid of the fundamental scheduling policies that have evolved over time to meet such practical goals. Implemented schedulers often contain heuristics developed through a combination of insights and trial-and-error that not have been theoretically analyzed or rigorously evaluated against each other, but that rather have worked well-enough in practice. Many supercomputer centers modify or replace vendor-supplied schedulers to meet complex goals [NAS, 2002; Feitelson and Jette, 1997; Lopez, 2002] concerning peak v. off-peak hours, different priority tasks, and debugging queues. An example is favoring tasks needing many servers because they have more difficulty obtaining resources to begin execution.

The literature on computing facilities often only briefly and superficially discusses scheduling policies. Documentation rarely justifies seemingly arbitrary scheduling policies although they often have a substantial effect on user experience (users dislike the long and variable times that their tasks wait in queue [Lopez, 2002]) and server revenue.

I concretize these thoughts by first examining the scheduling at supercomputer centers and then by cluster management software employed by IT resource providers and by first-generation computational grids. I finish this section with principles gleaned by abstracting away details and aspects outside of my thesis scope (Chapter 3) from these cases.

4.6.1 Supercomputer scheduling

A supercomputer center balances the following conflicting goals: maximizing throughput for shorter 'debugging' tasks and longer 'production' tasks, preventing any set of users from dominating the system, allowing a single user to dominate the system if approved by management, and enforcing the extant pricing mechanism, such as the service units (Chapter 4.2) allocated as the rewards of grants issued by funding sources. [NAS, 2002]

Such goals are often addressed with multiple queues corresponding to ranges of task service times (i.e., separate queues for small, medium, and large tasks) or task type (debug or production) and specific times of day in which long-running tasks are vacated to make room for tasks needing many nodes. The presumptive reason for different queues for different task size ranges is to reduce queuing delay; it is well known that task size variability leads to queuing, at least in open systems [Harchol-Balter, 2003b].

Thus users must specify memory and processor time requirements for queue placement [Feitelson and Jette, 1997]. Submitters often overestimate requirements because these systems often kill tasks whose estimates are exceeded. In any case, studies have shown that people are bad at estimating task size and that statistical techniques do better. Automatic task size estimation, while applicable to both non-speculative and batchactive scheduling, has not yet been widely deployed, and is discussed in Chapter 4.7.

The smallest queues are generally always available through PS for file editing, code compilation, and small test runs. For other queues, a more complex policy, some variant of FCFS or FB, is employed.

For example, at the NASA Advanced Supercomputer Center, tasks are sorted on priority. The highest priority task runs if resources are available, and if not, the system is drained for resources. For efficiency, tasks which can complete before the system is drained are back-filled [Feitelson and Jette, 1997]. 'Special' (as designated by an administrator) tasks are given the highest priority. Tasks with more parallelism have priority, and the system might be drained of tasks for them to run. There are several reasons for this: (1) it is more difficult for them to otherwise obtain resources, (2) they consume more resources and thus produce more revenue than smaller tasks, and (3) there is a perception that larger tasks do more important work (scientifically or otherwise).

Similar to user-FB (Chapter 4.5.3), long-waiting tasks obtain higher priority, and users that have recently (on the order of days) used a lot of resources have less priority for their tasks. This policy is a time- instead of resource-based version of decay-usage scheduling discussed below (Chapter 4.6.2). If a user does not have sufficient service units for running a task, then the task does not run. Additional heuristics, such as draining all tasks at the start of the non-primetime period of the day and restricting users to a certain number of queued tasks further ensure that tasks with high parallelism do not starve and that no small set of users dominate resources over others, respectively [NAS, 2002]. These complexities largely concern the scheduling of parallel tasks and of irregular tasks (such as non-production or administrator tasks).

There is a single queue used for all tasks on the Lemieux Supercomputer at the Pittsburgh Supercomputer Center (PSC) [Lemieux, 2003]. The task needing the most nodes runs first if the necessary number of nodes are available. Ties are broken by FCFS. At 5pm each day the system is drained of all tasks. Each task has a six hour limit (checkpointing is employed for those tasks requiring more time) and four queued tasks are allowed per user, avoiding starvation and resource abuse.

When tasks need only one server, i.e., when there is no task parallelism (which is a characteristic of the tasks in my thesis scope as described in Chapter 3.2), the NASA Advanced Supercomputer Center policy is a variant of FB and the Lemieux Supercomputer policy is a variant of FCFS. On the Lemieux Supercomputer, the preemption case makes the policy appear to long-running tasks to be PS with a time slice of six hours, which leads to better mean response time than strict FCFS, but not as good as FB for service time distributions with the DFR property (Chapter 4.5.1).

4.6.2 Cluster scheduling

When moving away from tightly-coupled parallel processing and removing non-production (debug) tasks, the policies at supercomputer centers and IT resource providers converge.

The main difference between cluster scheduling (employed on clusters and first-generation computational grids) and supercomputer scheduling is that users do not need to supply task resource requirement estimations (e.g., for processor time, memory, etc.) when submitting tasks to the former.

The most widely deployed cluster scheduling policy is decay-usage which is similar to the user-FB (Chapter 4.5.3) variant used in supercomputer scheduling (Chapter 4.6.1), except that it is resource- instead of time-based.

As in the supercomputer environment, the non-theoretical literature for deployed cluster and grid systems focuses on issues beside scheduling, assuming that a variant of FCFS or decay-usage will suffice. Recent theoretical work considers new and interesting approaches beyond these variants but which still assume that all tasks are non-speculative [Harchol-Balter, 2002].

Much research explores the availability of workstation idle time that could be harvested as cluster resources (Chapter 2.3.3), or how to support mixed interactive and parallel workloads on the same cluster [Arpaci et al., 1995]. The Globus Toolkit [Globus, 2003], a system for managing grids, and GLUnix [Ghormley et al., 1998], a research system for managing networks of workstations, focus on the following considerations (which are a subset of grid computing considerations): creating a single system image for transparent remote execution (including redirecting I/O and Unix signals among nodes), managing control and access to cluster resources, ensuring availability and failure recovery, increasing the speed of starting a large parallel task and other scalability issues, and enabling users to reserve specific machines with special devices or local storage when necessary. The scheduling in GLUnix was straightforward and not a research focus: tasks that were issued interactively would run on nodes with the fewest tasks recently in their scheduling queues; tasks that were issued with a batch submission mechanism would run FCFS as nodes became idle. A research clustering system called Sprite ambitiously supported transparent process migration to rebalance load when warranted by dynamic load conditions [Douglis and Ousterhout, 1991].

As a typical deployed cluster scheduling policy, Condor [2003], one of the most popular cluster management systems, strives to provide equal resource usage among users. The policy looks at the sum of the resource usage of all the tasks requested by a user and preferentially schedules tasks belonging to users whose tasks have used fewer resources, preventing a user from obtaining more than his 'fair' share by queuing large amounts of work. The amount of resources used by a user's tasks are decayed over time and the inverse ratios of these decayed values across users determines the ratios of resources the users will receive if they queue work, ensuring that tasks belonging to users whose tasks have used more resources do not starve. By default, resource usage is decayed exponentially with a half-life of one day. Moreover, tasks are preempted when resource usage among users with queued tasks becomes imbalanced using a one hour preemption granularity to prevent thrashing. Competing users will converge to an equal share of resources over time. This policy is known as decay-usage.

In decay-usage [Corbató et al., 1962], like user-FB, the user who has consumed the fewest resources is favored, but unlike strict user-FB, a user's resource usage is decayed over time. As resource usages decay, decay-usage behaves more like PS. The motivation is to favor interactive users — users who submit one task at a time and wait for task output before repeating — while not starving batch users — users who submit many tasks at once and who come back later to retrieve task outputs (e.g., those who submit tasks overnight). In other words, users using fewer resources get priority, while users using more resources are not penalized forever. Intuitively, long-past experiences should affect scheduling less than recent experiences. Typically using an exponential decay, resource usage older than some time is effectively forgotten.[19] Decay-usage usually also employs hysteresis to avoid thrashing, PS-like behavior between tasks of users who have used close to the same amount of resources.

Decay-usage, by definition, optimally minimizes the variance of (decayed) user resource usage, which is a primary scheduling goal, and as argued next (Chapter 4.6.3), does reasonably well among non-size-based policies at meeting the other goals in Table 4.2.

4.6.3 Summary of scheduling in practice

In practice, a variant of FCFS or decay-usage is employed for several reasons. A variant of FCFS might be employed because the user wants a clearly defined policy (such as 'FCFS, but preempt if a task runs continuously for six hours,' which is employed on the Lemieux Supercomputer) rather than having scheduling be dependent on resource usage. A user does not want his or her task to fluctuate in execution rate as the task uses more resources and as others' tasks execute. Delay is expected to be a function of queuing and then execution, not a function of other load once execution begins. Moreover, timesharing (as exhibited by many non-FCFS policies) is avoided in parallel contexts due to network communication among nodes, an application domain out of my scope. Timesharing is also avoided in non-parallel cases when memory preemption is an issue. These considerations lead to the use of schedulers like FCFS.

In other settings, such as in the Condor clustering system, users' batches are timeshared. Consider the traditional scheduling goals of minimizing mean response time, minimizing mean slowdown, and maximizing load (Table 4.2). The lineage of decay-usage from FB means that it does better, under the ubiquitous [Harchol-Balter and Downey, 1997] task size distribution with the DFR (decreasing failure rate) property, than other non-size-based policies at minimizing mean response time and mean slowdown (Chapters 4.5.1 and 4.5.2, and thus indirectly also does better than other non-size-based policies at maximizing load (Chapter 4.5.4). Further, when users issue ordered task sets, user-FB is likely to select tasks in the order of need across users. When resource usage is not directly charged, decay-usage is employed

[19] This is similar to the standard policy on a single Unix workstation elaborated on within Chapter 5.6.1: the multi-level feedback queue preferentially schedules tasks which wake after being blocked on other resources or user input over long-running, processor-bound tasks, while ensuring that those processor-bound tasks eventually run. The difference is that decay-usage is based up aggregate resources consumed by a user while a Unix multi-level feedback-queue looks only at the resources consumed by a task.

policy	description
FCFS	first-come-first-serve
user-FB	user who has used the fewest resources
SRPT	shortest-remaining-processing-time first

Table 4.3: Non-speculative scheduling policies (compare to speculative task policies in Table 5.3).

to prevent resource abuse. When resource usage is directly charged to the user, preventing abuse via scheduling is not necessary.

Batchactive scheduling is compatible with both deployed approaches and the results of this thesis look at speculative extensions to FCFS, user-FB, and SRPT (Chapter 6.2). These policies are summarized in Table 4.3.

4.7 Predicting task service time

Scheduling decisions sometimes require knowing the service time of individual tasks. Often, the scheduler does not have this information and users cannot be expected to provide this data reliably. The alternative is for the scheduler to automatically estimate task service time. The problem of learning task resource 'footprints' is undergoing strong study by many researchers.

One use of task size information is for applying SRPT for non-speculative scheduling or SRPT as a building block of a batchactive scheduler (Chapter 5.5.4). Doing so improves both to the same extent as shown in Chapters 6.2.4 and 6.2.8. Note that predicting task size is not necessary for batchactive scheduling; it is only to obtain additional performance equally gained by non-speculative schedulers.

Besides its use for scheduling, service time predictions along with what percentage of their service time tasks have already consumed can help a person plan his or her day, and can relieve a person's anxiety of waiting for task output.

Across diverse tasks, service time can vary widely; its coefficient of variation (Chapter 4.7) has been measured to range from four to 70 at several supercomputer centers [Feitelson et al., 1997]. Moreover, Feitelson and Jette [1997] observed that people are bad at estimating service time. Smith and Wong [2002] showed that automated service time prediction is significantly more accurate than human predictions.[20]

[20] Still, many supercomputer centers ask the user for this estimate, both to decide in what queue a task should be placed and to kill a task that exceeds its limit. [NAS, 2002].

The approaches can be classified as basing predictions on task characteristics or on task sizes of recently executed tasks. (Sometimes a hybrid is employed [Narayanan et al., 2000].) I detail the former approach in the bulk of this section and I sketch the latter at the end. In both cases, measurements of executed task service time are made. Under preemptive scheduling, clock time is not an accurate measurement of task size; processing (virtual) time is easy to measure on Unix variants to determine a task's actual processing needs.

Service time prediction based on task characteristics depends on the assumption that a task run twice on the same inputs will have the same service time.[21] Task inputs are classified as discrete or continuous parameters; e.g., a command-line option that turns on or off some feature or a command-line option that takes a real value as an argument, respectively. The size of an input file used by a task is also a continuous parameter. Spring and Wolski [1998] show that the service time of the `complib` biological sequencing library is highly predictable from input size. The service time for the BLAST DNA similarity searcher is dependent on the sizes of the sequences under comparison and the search accuracy; the accuracy is determined by the choice of search algorithm, a discrete parameter. In computer rendering, once a shot (scene) is being processed, the runtimes of individual tasks are 'generally predictable' [Epps, 2004]. Table 4.4 shows how service time is a function of task parameters from such and other tasks.

Other researchers and I have used curve-fitting to predict service time from such parameters with high accuracy.

Service time can be predicted after learning the relationship of task parameters to past service time. A predictor can evaluate polynomials of the form $t = c_0 + \sum_{j=1}^{n} c_j x_j$, where t is the service time of a task, the xs are functions of the task's continuous (knob-like) parameters (like input size), and the cs are coefficients set by a linear least squares regression [Gauss, 1821] to best fit past observations. Constructing these polynomials involves trial and error; the number of terms balances expressiveness with overfitting. For discrete (switch-like) parameters (like the type of algorithm employed by a DNA matching algorithm), additional sets of coefficients, one per each discrete parameter setting, should be learned. Examples of parameters were shown in Table 4.4.

I built a service time predictor using the linear least squares regression code from LAPACK [Anderson et al., 1999]. To demonstrate the concept, I

[21] Cross-invocation caching of state, such as cross-invocation dynamic programming, may violate this assumption.

problem	task service time	service time factors
find DNA sequences	seconds to hours	source × target DNA size; search algorithm
apply lighting to scenes (radiosity)	minutes	number of polygons in a scene
optimize data placement	≈1 hour	number of accesses
find file cache access patterns	several hours	number of accesses
find file access relationships	30–60 minutes	number of accesses
study MEMS scheduling	seconds to hours	number of accesses
study microarchitecture	4–24 hours	number of instructions
study computer virus propagation	20 seconds to 6 hours	network topology, virus birth and death rates
evaluate brain state models	≈ 10 hours	(unknown)

Table 4.4: Evidence that many tasks have predictable service times. Sources: local survey, personal communications, anecdotal experience. [ECE, 2002; Pereira, 2003; Wenisch, 2003; Narayanan et al., 2000; Biowulf, 2004]

looked at the resource usage of a program called Radiator which applies a radiosity algorithm to static images of three-dimensional scenes. The technique is applicable to predicting runtime on an otherwise unloaded machine, a machine with non-preemptive scheduling, or a machine with preemptive scheduling given an appropriate server model that considers the effects of preemption, e.g.

Radiator applies radiosity algorithms to make scenes look realistic, and its runtime is a function of the number of polygons in a scene (a continuous parameter) and the radiosity algorithm employed (a discrete parameter) [Willmott, 2000]. I used 5, 448 test Radiator runs from a 230 MHz Pentium MMX gathered for multi-fidelity research by a colleague [Narayanan et al., 2000]. My predictor learned how Radiator's parameters map to the demand of processing cycles by finding the constants in $c_0 + c_1 p \log p + c_2 p^2$, where p is the number of polygons used in the scene. This polynomial reflects the complexity of the radiosity algorithms. Figure 4.7 shows how the resource demand of the task depends on the input scene and the number of polygons used in the radiosity computations. It is feasible to apply this regression online: The time to regress over 5, 528 elements occupying 130 KB on a 700 MHz Pentium III was 4.564 ms averaged over ten trials with a standard error of 0.021 ms. This overhead would be even lower if I had used a LAPACK library optimized for this architecture.

Ideally, the predictor would not need many samples to make accurate predictions. After training on $i - 1$ samples, I had the predictor predict

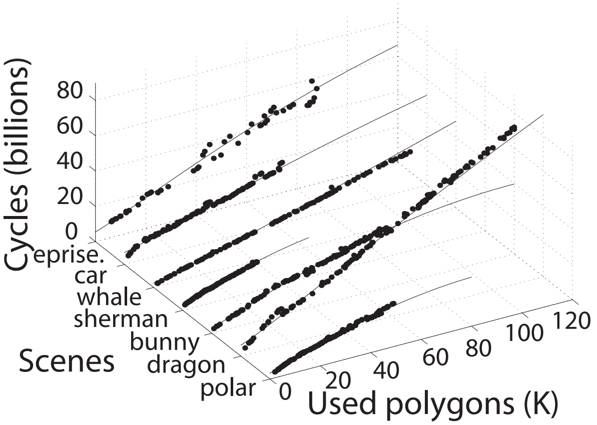


Figure 4.7: An example of applying regression to predict service time. A radiosity task consumes different amounts of processor resources (cycles) for different scene models and different numbers of polygons for each model. Points indicate samples of past runs. Curves indicate functions fit to these points; these functions would be evaluated to predict resource needs for a scene at a specific number of polygons even if that number has never been run before. Service time can be predicted with this same concept.

the resource demand of the ith run of a task and then measured the error (the normalized difference between the prediction and subsequent reality) as shown in Figure 4.8. Two types of tasks and resources were explored. Because the polynomials fed to the regression used had three coefficients, I ran three initializing trials before taking numbers (this is a requirement for linear regression). The maximum error was about 20% at the start of the radiosity algorithm, and after that, most error was around 5%. To my knowledge, the extent to which a scheduling policy, such as SRPT, can tolerate service time prediction error before a non-size-based policy would perform better has not been studied.

There exists much work detailing successful predictions from task parameters. Kapadia et al. [1999] use regression to predict the resource requirements of tasks over their parameters. These predictions are used for allocating resources on a computational grid. Abdelzaher [2000] introduced a profiling subsystem using regression to predict the demand of an Apache web server serving both static and dynamic web pages. Narayanan et al. [2000] use regression to predict the resource demands of mobile applications. Narayanan [2002, ch. 6] describes prediction techniques in detail. To estimate the selectivity (related to demand) of a database query, Chen and

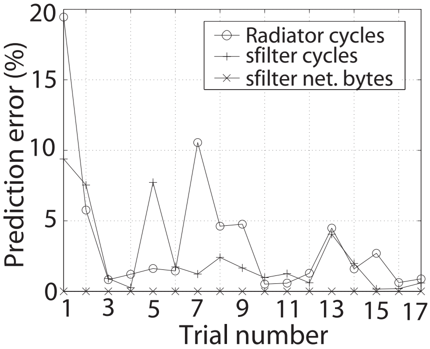


Figure 4.8: How prediction error decreases with more task runs. The more service time data that the regression has, the more accurate its predictions are. The maximum error was about 20% at the start of Radiator, and after that, most error was around 5%. Another application called `sfilter` (described in Petrou [2002]) is also shown.

Roussopoulos [1994] apply a *recursive* least squares regression, incorporating online samples faster and with less memory overhead than a normal regression.

Turning away from regression, service time prediction based on recently executed tasks depends on the assumption that recent past is a good predictor of near future. An effective approach is to take an average of past task service times and weigh more recently executed tasks more heavily, a technique known as an exponentially weighted moving average (EWMA). Kim and Noble [2001] note that many past attempts at passively estimating network bandwidth rely on EWMA filters with static factors for weighing the present over the past, resulting in filters that are either stable (in light of noise or temporary transients) or agile (in response to new behavior), but not both. They tested four dynamic filters against two static filters and recommend one of them, a *flip-flop filter,* which uses techniques from statistical process control.

A discussion of other statistical or machine learning techniques for predicting service time is outside my scope. I chose to present least-squares regression and EWMA because the attention many researchers have given them attests to their usefulness for this purpose.

4.8 Inadequacies when speculative tasks are present

The theoretical and deployed scheduling policies described in this chapter were designed with non-speculative tasks in mind; that every submitted task was equally known to be needed by the users who submit them. However, there exist important application scenarios (Chapter 2.2) in which this is false. Problems arise when users speculate with existing schedulers. There is confusion as to how many tasks a user should submit, resulting in ineffective scheduling (poor time- and cost-based scheduling metrics). The problems cannot be overcome without a scheduler that can discriminate between speculative and non-speculative tasks, i.e., a batchactive scheduler.

Should a user exploring a space speculatively submit one speculative task, a few, many, or the entire 'computational plan?' When resources cost, the user is pressured to only submit a few tasks at once (at the limit, to behave interactively), because the user does not wish to be charged for running tasks whose outputs are unneeded. But doing so leads to poor time-based scheduling metrics because the execution of speculative tasks is not fully pipelined with the user's think time of completed tasks. When resource usage is not directly charged, or if the user is willing to pay for unneeded tasks, then the user should submit many speculative tasks (at the limit, to behave in a batch manner) so that the scheduler can execute them before needed. (See Figures 1.3 and 2.1). However, if every user did this, then resources would be overwhelmed with speculative tasks and the response times for non-speculative tasks would increase dramatically. (These implications of speculative user behavior on non-speculative scheduling policies is detailed in Chapter 6.2.)

Users are in a double-bind: to appear to be good citizens, users would like to appear to submit a 'reasonable' number of speculative tasks, in hopes of balancing their wasted costs and visible response time with the visible response time of other users. However, a user who does not submit swaths of speculative tasks when resource usage is not directly charged or is affordable will secretly be regarded as a simpleton by his or her peers for not abusing the system. The problem is deeper, however. Even if everyone wished to cooperate, there is no clear way to compute the correct number of speculative tasks to submit. Whether or not submitting a speculative task that turns out to be needed will help a user's visible response time or whether submitting a speculative task that turns out to not be needed will hurt others' visible response time or cost the submitting user significantly depends on many factors unknown to the user: the task arrival process, task service time, user think time, and the probabilities that speculative tasks will be needed.

As mentioned above (Chapters 4.6.1 and 4.6.2), the scheduling policies for supercomputers and for clusters have been given short analytic shrift. Beyond the lack of analysis or detailed justification for the policies, there is a lack of attention given to scheduling speculative tasks.

As detailed later (Chapter 5), speculative user behavior introduces new challenges. When a user speculatively issues tasks, there is a notion of task deadlines; the time that a speculative task might be needed which occurs later due to user think time (Chapter 2.3.1) and away periods (Chapter 2.3.2). Thus two traditionally separate scheduling worlds converge: response time-based and deadline-based. It has been suggested that scheduling analysis becomes harder when tasks are needed sometime after submission and that a consideration of this important user behavior is overdue; that scheduling theory and practice have diverged [Feitelson et al., 1997].

4.9 Summary

This chapter presented the schedulers against which I compare my batchactive schedulers. These schedulers do not treat speculative tasks as first-class entities; to them, all submitted tasks are equally known to be needed by the users who submit them.

To provide this scheduling background, I described the architecture under consideration: clusters and computational grids. I argued that this architecture is prevalent and important. I described how computational resources may be outsourced or may be used by those who own them or are affiliated with those who own them. These two relevant relations lead to two situations under study: one in which resource time is charged to the resource user (a profit-center) and one in which resource usage is not directly charged (a cost-center), respectively.

Then I formalized the interaction among users, tasks, and servers which led to the definition of important scheduling metrics (Table 4.1). In terms of utility, I derived user and resource provider scheduling goals based on these metrics, including minimizing mean response time and maximizing load (Table 4.2).

I described fundamental scheduling policies with respect to how they achieve such scheduling goals. Then I looked at typical policies employed in supercomputer and cluster settings and derived their lineage from the fundamental policies. By ignoring heuristics which favor tasks with a high degree of parallelism (because such tasks are outside my scope as discussed in Chapter 3.2), it became apparent that scheduling in practice is either a variant of FCFS or decay-usage. This finding allows me to have practical policies to act as baselines against which I quantitatively compare batchactive scheduling (Chapter 6.2). As an aside, I showed how task service time can be predicted so that better, size-based scheduling, can be employed.

I ended this chapter discussing shortcomings of non-speculative schedulers when presented with speculative workloads. Users cannot know how deeply to submit speculative tasks, leading to worse scheduling metrics. By discriminating between speculative and non-speculative tasks, my results (Chapter 6.2) show that batchactive scheduling (Chapter 5) provides a better computing experience, in terms of time- and cost-based scheduling metrics, than common practice.

The ability to see what has to be done and know the exact last possible minute that it can be done is a very heavy burden indeed.

Sandra Thompson, *Art of Improvisation*

5 Batchactive scheduling

I present *batchactive scheduling* to reduce or eliminate visible response time (the time a user is 'blocked on' task output), among improving other scheduling metrics, across people or autonomous agents executing speculative tasks — tasks that at the time of submission the submitter does not yet know if they will be eventually needed [DeGroot, 1990]. I extend the definitions, metrics, and goals presented in the chapter on non-speculative scheduling (Chapter 4). Similar aspects between scheduling environments are covered quickly and differences are elaborated on.

A task could be marked speculative by its submitter and assigned a probability of eventual need. A user could assign a real between 0 and 1 indicating the probability that he or she thinks at the time of submission that the task would be needed, possibility to be updated later. Such data, if accurate and if used correctly, could lead to better scheduling. However, it is unrealistic for users to provide such probabilities; it is a burden and accuracy cannot be assumed. Dawes [1979] demonstrated that the more choices presented to a person, the less likely that he or she would use them effectively.

Instead, in a batchactive scheduler, as I define it, users tag tasks as either speculative or needed; there are no 'levels' of speculation. Users *disclose* tasks whose outputs they are not sure they will need at that time, i.e., the speculative tasks, and *request* tasks whose outputs they already know they need. Later, a user may promote a disclosed task to a requested task, or a user may cancel any task.[1]

Task disclosure is a hint. Hinting is a common and longstanding technique to improve system performance [Patterson III, 1997]. Lampson [1983] reports the use of hints in operating systems, networking, and languages. His hinting examples use potentially out-of-date information to short-circuit expensive computations. Another type of hint expresses policy advice from one

[1] For reasons of accounting mentioned in Chapter 5.2, a requested task may not be later classified as disclosed.

system component to another, such as an application choosing among file cache policies [Cao et al., 1994]. The mechanism of conveying that a task is speculative is a 'disclosure' hint in the terminology of Patterson III [1997] and exists in the context of several I/O systems [Kotz, 1997; Parsons et al., 1997; Patterson et al., 1995; Steere, 1997]. Disclosure hints differ from hints which give advice. For example, an advising hint might specify that a speculative task should have less priority than a needed task. Advice hints uses one's knowledge of application behavior, system resources, and system implementation to declare how resources should be managed. They are brittle and break modularity. Disclosure hints only reveal user knowledge, enabling the system to globally optimize resource management. Further, they express information independent of system implementation; viz., they remain correct when the application execution environment changes. Finally, the disclosure interface, being the same as for requesting non-speculative tasks, should be easy to use.

People wish to batch their planning and submission of tasks and pipeline the consideration of completed tasks with the execution of remaining tasks (Figure 1.1). Non-speculative schedulers present obstacles to this way of working (Chapter 4.8). With batchactive schedulers, users are free with a novel batchactive pricing mechanism to disclose speculative tasks, knowing that tasks will be ordered intelligently, based in part on whether or not they are speculative, to improve important scheduling metrics. Most of the time, this is accomplished by giving requested tasks precedence over disclosed tasks.

To gain the benefits of batchactive scheduling, the system requires users to disclose their computational plans; this work makes no attempt to guess what tasks are more or less useful to the user. The application scenarios in Chapter 2.2 show that there exist important problems in which tasks can be categorized as either speculative or needed.

Endowing servers with the knowledge of tasks that may be needed in the future enables the servers to get an early start rather than being idle. Further, this knowledge can expose parallelism within a user's workload that the scheduler can use to leverage multiple cluster nodes simultaneously when tasks do not depend on outputs from one another. Such 'parallel searches' evaluate a number of alternative execution paths simultaneously as opposed to those which consist of a linear execution sequence with data dependencies [DeGroot, 1990].

Batchactive scheduling motivates users to distinguish between speculative and needed tasks and to disclose speculative tasks deeply. Users will observe lower mean time waiting for task output, making them more pro-

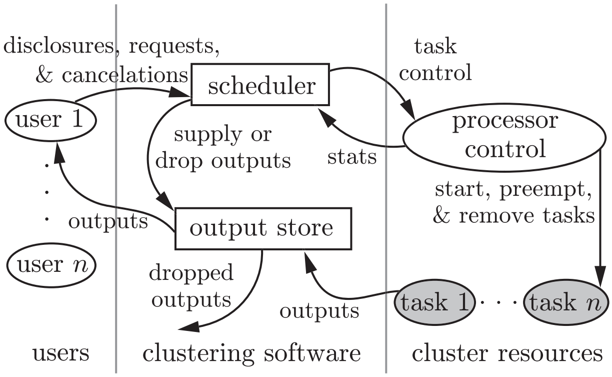


Figure 5.1: Interaction between users, the batchactive clustering software, and the cluster resources. (Compare to the non-speculative scheduling version depicted in Figure 4.1.)

ductive and less frustrated. The effect on scheduling metrics exhibited by batchactive scheduling is quantified by the simulation results in Chapter 6.2.

Figure 5.1 depicts user interaction with batchactive clustering software and the software's interaction with the cluster resources. Any number of users disclose, request, and cancel any number of tasks. The scheduling policy decides which and when disclosed and requested tasks run. If a disclosed or requested task is canceled, it is no longer a candidate. The scheduler communicates decisions to the operating systems running on the cluster resources which handles the details of running tasks on the servers and provides task statistics to the scheduling policy, such as how long a task took to run. If a task executes and was requested, then the task's output is supplied to the requesting user. If a task executes and was disclosed but not requested, then the task's output is stored in a location isolated from the rest of the system until requested or canceled.

The cluster architecture for batchactive scheduling is the same as that for non-speculative scheduling (Chapter 4.1), and thus is not discussed in this chapter.

It is well-known that not every I/O is equal: those I/Os that block a program's progress matter more to the user experience than those for prefetching, e.g. [Ganger and Patt, 1998] Likewise not all tasks are equal. Those tasks that a user is waiting on are more important than others. How a scheduler should treat such tasks differently is the subject of this chapter.

First I discuss how the cost model, definitions and metrics, and scheduling goals differ from common practice (Chapters 4.2, 4.3, and 4.4) when the scheduler is made aware of speculative tasks. Then I describe batchactive scheduling policies that utilize difficult to obtain information followed by easier to deploy batchactive policies implemented and studied in simulation. I discuss ways in which existing schedulers may be transformed to be batchactive. I continue with a discussion of how certain scheduling inputs, when they are not known, can be predicted for the policies that need them. Following is a discussion of some ways in which users may attempt (and often ultimately fail) to abuse the batchactive pricing mechanism and scheduling policies. Before summarizing this chapter, I consider how speculative scheduling can operate beyond centrally scheduled processor resources.

5.1 Batchactive cost model

As in the non-speculative scheduling environment (Chapter 4.2), I consider two relations between resource owner and resource user: that (1) they are the same self-interested entity (e.g., a laboratory or university cluster) and resource usage is not directly charged to the user; and (2) they are separate entities and resource usage is charged. These are the cost- and profit-center models, respectively. This section details a new pricing mechanism for speculative tasks. The *batchactive pricing mechanism* encourages users to disclose speculative work deeply, enabling the scheduler to best meet scheduling goals.

5.1.1 Problem with the non-speculative pricing mechanism

Users might hesitate disclosing tasks if there were a risk they would be charged needlessly, as they would under the pricing mechanism for non-speculative scheduling (Chapter 4.2) which charges for all resource usage. A user would tend to not disclose work that had a small chance of being needed. This is problematic, because batchactive scheduling policies work better with more information.

Further, users might hesitate submitting speculative tasks even if there were a good chance that they would need such tasks. The following digression elaborates on this characteristic of human behavior.

Kahneman and Tversky [1979] developed prospect theory which seeks to model the irrationality of human decision-making; i.e., how people form judgments and make choices. The prevailing assumption, at least to support analytic tractability, was that beliefs and decisions conformed to logical rules and that people acted as 'rational agents' to increase their utility. These researchers instead showed that some behavior is systematically illogical.

Among other findings, Kahneman and Tversky showed that people feel a reduction in well-being is more acutely than increases, and that people underweigh probable outcomes compared to certain outcomes. This 'certainty effect' explains an observed asymmetry between risk-averse and risk-seeking behavior.

This irrationality translates to the batchactive environment. Even when it is probable that a user would gain by disclosing a task (because the user believes, with say 90% confidence, that he or she will need its output), the user would tend to avoid the risk in being charged for a task not certain to be required.

The batchactive pricing mechanism described next (Chapter 5.1.2) takes the risk out of task disclosure for both unlikely and likely needed tasks. This change is intended to motivate users to submit speculative work.

5.1.2 A new pricing mechanism

Traditionally, computing centers charge for resource usage irrespective of whether tasks were needed [Lemieux, 2003]. The batchactive pricing mechanism for speculative scheduling diverges from this norm.

Requested (needed) tasks are priced as usual (Chapter 4.2). This includes tasks originally requested and tasks originally disclosed as speculative but later learned to be needed by the user.

However, disclosed tasks that were never needed are not charged. No charge is levied irrespective of whether such a task did not run at all, ran for part of its service time, or ran to completion. That is, the batchactive pricing mechanism charges for resources used only by tasks whose outputs are requested. The resources used by a requested task are charged even if those resources were consumed before the task was requested; i.e., while a speculative, disclosed task had not yet been requested. Thus, a resource provider's revenue is proportional to the amount of requested work during some time period. This relationship between requested load and server utility is depicted in Figure 5.2.

With this mechanism, the user would not need to weigh the cost (wasted money) and benefit (better response time, which would only be a guess, based on scheduling policy and server load) of each disclosure, allowing the user to freely disclose tasks. This small but radical change in pricing mechanism has numerous implications as described in the next section (Chapter 5.1.3).

It is important to know whether this *pricing mechanism* leads to a rational *user strategy* (even if users do not always behave rationally, as mentioned

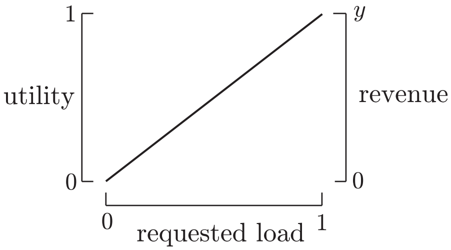


Figure 5.2: How requested load affects server utility and revenue under the batchactive pricing mechanism. (Compare to Figure 4.3.) Requested load, the fraction of time a resource is busy with originally or eventually requested tasks (i.e., ignoring speculative tasks eventually known to not be needed) varies between 0 and 1. Server utility or revenue is tied to the amount of resources a server charges. Under the batchactive pricing mechanism, only requested tasks are charged. As requested load increases, utility increases equally. Revenue can be calculated directly from requested load given a constant cost per unit resource. Shown at maximum utility (1) is maximum server revenue (y).

in Chapter 5.1.1). Under the non-speculative scheduling pricing mechanism (Chapter 4.2), the user strategy is simple: At the time that a user determines that requesting (and paying) for a task will likely increase his or her utility, then the user should request the task. In the speculative scheduling pricing mechanism, there is also the option of disclosure. If a user does not know whether seeing a task's outputs would increase his or her utility, then the user should disclose the task. Consider time discretized. At each time step after disclosure, the user may either request the task or do nothing. At the time that a user determines that seeing the output of the disclosed task would increase his or her utility, then the user should request the task. Until that time, the user may safely wait; he is not charged by indefinite waiting. (Disclosed tasks may be reordered or canceled if the user determines that the output from one is more likely to be needed than another.)

5.1.3 Consequences

The batchactive pricing mechanism is beneficial from the user's perspective because speculative work can be submitted at no cost. However, it is not obvious whether a resource owner would prefer this pricing mechanism. Compared to the traditional practice of charging for all resource usage (Chapter 4.2), the resource owner could lose revenue from processing time consumed by disclosed tasks that were never requested.[2] The improved visible response time of batchactive scheduling might compensate: as users spend less time waiting for task output, they submit needed, chargeable work more quickly. Even if users do not submit needed work more quickly, the batchactive pricing mechanism may still benefit the resource provider as discussed next. Note that the apparent additional billed load from the traditional pricing mechanism may not occur in practice: users with batches of work would likely resist deep disclosure because they do not wish to be charged for needless speculation, and even if they were willing to pay, the resulting visible response times would be poor.

In a cost-center, if the batchactive pricing mechanism results in lower revenue, the price of requests can be raised to make up the difference. Users with batches of work will pay less because they are not charged for unneeded speculation. Users with only a few tasks outstanding at any time will pay more, as they are paying for the canceled tasks of others. (Recall that, actually, costs in a cost-center are usually not directly charged to the user, but to an aggregate part of the organization.) Both kinds of users will receive better value in the form of lower mean visible response time, as explained in Chapter 6.2.2, while the billing total remains unchanged.

A profit-center can be motivated to use the batchactive pricing mechanism because the significant value provided with batchactive scheduling could encourage additional users, deeper speculation, and bigger tasks, any of which would raise server revenue. Results show (Chapter 6.2) that at any level of billed load, batchactive scheduling provides better mean visible response time. Latency-sensitive users will not push traditional schedulers into regions of high billed load because, at those levels of revenue, visible response times are too high. The latency threshold for batchactive scheduling, in contrast, is better. In other words, because of the better scheduling offered by batchactive policies, more users can be supported with less degradation of time-based metrics. Users might switch to computing centers that charge for only requested tasks and provide better visible response time, and thus the owners of such centers might profit more because of the greater demand.

I study these considerations under the assumption that the marginal cost for a provider's resources to do work rather than be idle is insignificant. If instead a busy server, due to heating costs, current draw, etc., costs significantly more to operate than an idle processor, then adjustments can be made to the results in Chapter 6.2 to determine resource revenue.

These issues are discussed further in Chapters 6.2.2 and 6.2.5.

[2] I do not advocate charging a reduced amount for unneeded speculative work because this could dissuade disclosure.

5.1.4 Dismissed extension for selling completed speculative tasks

The batchactive pricing mechanism is simple to understand and lends itself to a simple task submission strategy. The mechanism can lead to at least one state, however, in which it seems that additional efficiency (improved utility for both users and resource providers) can be obtained. I describe this state and why it is hard to obtain this efficiency with an extension to the pricing mechanism.

Consider a user who has submitted a speculative task but who never requests its output. Suppose that a batchactive scheduling policy runs this task to completion. For some period of time, the user has not requested this task, indicating that the user has not determined that paying the advertised price (the product of task size and a known price per unit resource) for this task is in his or her interest.

Consider further that the user attaches some non-zero value (but less than full price) to this task. The resource provider, with the completed task output, would benefit from selling the output for less than full price. In fact, the resource provider would benefit from selling for any amount more than zero.

The problem is determining a price for this task when the resource provider does not know the value attached to the task by the potential buyer (a case of asymmetric information). A further complication is that no other user would be interested in purchasing the submitter's task output; the resource provider cannot sell to the highest bidder, because in this degenerate market there is only one potential buyer.

Any mechanism in which the resource provider elicits information from the user, such as 'will you buy this completed speculative task's output for x dollars, where x is less than the advertised price?' or 'please specify a function conveying the decreased value of older task output' can be abused. A user can underreport his or her value of the task.

One way for the resource provider to pressure the user to more accurately bid for such task output exists in an iterative setting: if the user pays significantly less than the advertised price of a requested task, then the resource provider may occasionally refuse to sell a particular task's output for some penalty time period. However, how to define how often and under what circumstances this should occur so that the buyer and seller converge to mutually beneficial prices is difficult.

I consider this pricing extension for selling unrequested, disclosed tasks outside my scope. It has parameters difficult to set, may be abused, and does not lend itself to a simple user strategy. I advocate the mechanism described above (Chapter 5.1.2) knowing that some efficiency is lost: The user pays for all the resources consumed by requested tasks. The user does not pay for disclosed, speculative tasks that are never requested.

5.1.5 Summary of the batchactive cost model

This section described the batchactive pricing mechanism.

A user would prefer not being charged for disclosed but never requested tasks. The more knowledge that the scheduler has of a user's computational plan, the better it can order tasks to meet important scheduling goals. To motivate users to disclose freely and deeply for tasks likely and unlikely to be needed, only requested resources are billed. The implications that this mechanism has on scheduling goals is described in Chapter 6.2.2. A discussion of how users might attempt to abuse resources under this mechanism in conjunction with the forthcoming batchactive scheduling policies (Chapter 5.5) is presented in Chapter 5.8.

5.2 Batchactive definitions and metrics

I extend the non-speculative scheduling definitions and metrics from Chapter 4.3. Analogous definitions and metrics are covered quickly and differences emphasized. The batchactive definitions express the notion of a user disclosing ordered or unordered tasks that might be requested in the future. With this information, batchactive policies (Chapter 5.4 and 5.5) can 'prefetch' task output. New metrics are needed to measure the response time of tasks that might begin running before being requested. In speculatively doing work, batchactive policies might run disclosed tasks that are never requested. To account for these resources, which can affect user costs and server revenue in light of the batchactive pricing mechanism (Chapter 5.1), additional metrics are introduced. The metrics in this section are referred to in Chapter 5.3 to define scheduling goals for a scheduler of speculative tasks.

There are *servers, users,* and *tasks.* Users arrive and depart.

A task can be disclosed, requested, canceled, executed, or finished. The disclosed and finished states are new. A task cannot simultaneously be in more than one state, except that a task can be both executed and finished. As in Chapter 4.3, I refer to the set of all tasks as A.

An arrived user can disclose one or more tasks. The set of *disclosed tasks* from all users up to time t is denoted by $A^d(t)$.

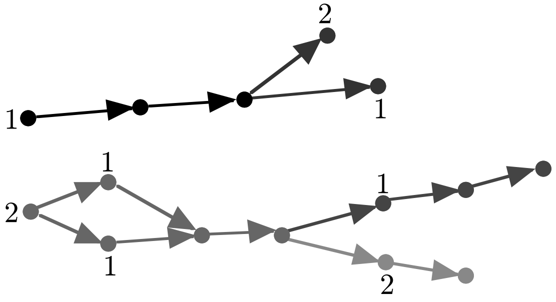


Figure 5.3: A task set composed of a weighted DAG of tasks in which later work (toward the right) is often more speculative. Connected components and branches are assigned numerical priority; equal numbers indicate no ordering preference. Different shades of grey represent different lines of inquiry that the task set submitter wishes to explore.

The set of disclosed tasks from one user is called the user's *task set.* In general, a user could specify as weighted directed acyclic graphs (DAGs) the order in which disclosed tasks should run, as shown in Figure 5.3. The DAG's connected components and branches within the connected components could represent subproblems, independent hypotheses, separate lines of inquiry; or simply the input / output dependencies across tasks. A user may reprioritize tasks and branches of tasks in his or her task set DAG. Clustering software for conveying DAG task execution order exists in the context of non-speculative scheduling; an example is Condor DAGMan [2004].

Determining and maintaining DAG orderings is likely tedious. Besides arbitrary DAGs, tasks may be desired in a flat list order or with no ordering preference, as shown in Figure 5.4. The list represents users who know that task A should be done before task B, which should be done before task C, etc., reflecting an any-time or iterative improvement task set (Chapter 2.1) or sequential tasks (Chapter 2.2.2). Tasks listed later are likely more speculative. Although not shown as a user command for simplicity in Figure 5.1, a user may reorder tasks in this list. The unordered collection, often used when randomly sampling a large space, indicates that the user does not mind which task outputs are returned first; any answer is helpful until more is known about the space, reflecting parameter studies (Chapter 2.2.3). Unordered tasks in the domain of I/O were called 'dynamic sets' by Steere [1997] (Chapter 2.5.3), and enabled the I/O system to return requests in the fastest order. I have only studied the simple case of flat list order in my results (Chapter 6) because the marginal value for complex orderings is not obvious.

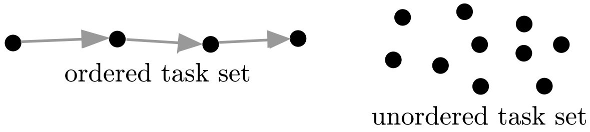


Figure 5.4: Two typical task set organizations: flat list and unordered. In the list, tasks to the right are more speculative.

A scheduling policy needs to know which disclosed tasks are candidates for execution. These tasks include all disclosed tasks from unordered task sets and the unexecuted tasks ordered first among all users' ordered task sets (i.e., due to DAG or list constraints). These tasks are denoted at time t by $A^{\bar{d}}(t)$, where $A^{\bar{d}}(t) \subset A^d(t)$.

An arrived user can request one or more tasks. Let t_a^r denote when task a was requested. The set of *requested tasks* among all users up to time t is denoted by $A^r(t)$. A requested task becomes no longer disclosed had it been already disclosed. That is, if $a \in A^r(t)$, then $a \notin A^d(t)$. A requested task cannot be made disclosed by a user, whether or not it began as disclosed because then it becomes unclear how much the user should be charged (Chapter 5.1): while requested, according to a batchactive policy it may have been favorably scheduled with respect to other disclosed work, implying that it should be charged, yet if it's output is never requested by the time the task completes, then its output is never seen, implying that it should not be charged.

An arrived user can cancel tasks that he or she had previously requested but that have not executed to completion. Disclosed (but not yet requested) tasks, whether or not they have executed to completion, may be canceled because disclosed task outputs are not immediately given to a user when a task completes, as explained later in this section. Also, the system cancels tasks associated with departed users. A canceled task remains forever canceled. The set of *canceled tasks* up to time t is denoted by $A^c(t)$. A canceled task becomes no longer disclosed, requested, or executed, whichever it was before cancelation. That is, if $a \in A^c(t)$, then $a \notin A^d(t)$, $a \notin A^r(t)$, and $a \notin A^e(t)$. The applications within my scope do not require the rollback mechanisms found in other work after task cancelation (Chapter 3.1).

Each task $a \in A$ has a corresponding *service time* S_a and *resource usage* $r_a(t)$.

Disclosed tasks ordered first, unordered disclosed tasks, and requested tasks are candidates for the *scheduler* to choose. A task runs at time t if and only if the scheduler decided so. The set of *running tasks* is denoted by

$$A^{\star}(t) \overset{\text{def}}{=} \big\{a \in A^{\bar{d}}(t) \cup A^{r}(t) \mid a \text{ runs at time } t\big\}.$$

A task's resource usage starts at 0 when it enters the system and increases by the amount of time that it runs.

Let t_a^e denote the time that r_a grows to equal S_a. The task is considered from this time on to be executed. The set of *executed tasks* up to time t is denoted by

$$A^{e}(t) \overset{\text{def}}{=} \{a \in A \mid r_a(t) = S_a\}\,.$$

The way that the batchactive scheduling environment controls access to task output differs from the non-speculative environment: If a task becomes executed at some time t and the task was disclosed, the scheduler removes the task from the set of disclosed tasks. That is, if $a \in A^e(t)$, then $a \notin A^d(t)$. At this point, the computing system stores the task's output in an isolated location until the task is requested or canceled. If canceled, its output is removed from the system. If a task is executed and requested, then the task is considered from the time that both occur and on also to be finished. The set of *finished tasks* at time t is denoted by $A^f(t)$. When a task becomes finished, the computing system provides the task's output to the requesting user, and the scheduler removes the task from the set of requested tasks. That is, if $a \in A^f(t)$, then $a \notin A^r(t)$.

See Figure 5.5 for a pictorial representation of the states in which a task can reside in batchactive scheduling.

Now I describe batchactive scheduling metrics which are used to evaluate how well the scheduler achieves the scheduling goals described in Chapter 5.3.

Mean *visible response time* is the main metric under consideration. A task with service time S_a is needed by the user at time t_n and executes (completes) at time t_e. A requested and executed task a has a corresponding visible response time defined as

$$V_a^{\text{resp}} \overset{\text{def}}{=} \begin{cases} 0 & \text{if } t_n > t_e, \\ t_e - t_n & \text{if } t_n \le t_e. \end{cases}$$

This consideration of when a task was needed, instead of when a task was requested (by users behaving speculatively with a non-speculative scheduler) or disclosed (by users with a speculative scheduler), is novel and more useful than the well-known definition of response time reviewed in Chapter 4.3 which conflates the time a task was requested with the time a task

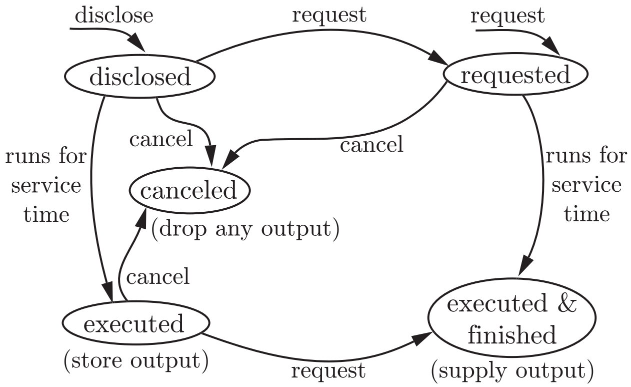


Figure 5.5: Batchactive task state transitions (compare to Figure 4.4). When a task's resource usage equals its service time, the task becomes executed. If a task is both executed and requested, then the task is considered finished and the task's output is supplied to the requesting user. If a task executes and was disclosed but not requested, then the task's output is stored in an isolated location until requested or canceled. If the task is canceled after executing, its output is dropped. Disclosed and requested tasks may also be canceled.

was needed. I suspect, along with Feitelson et al. [1997], as elaborated in Chapter 5.4.1, that this overloading has subsisted because of the difficulty in knowing when speculative task outputs are needed by a user.

This definition of visible response time overestimates when a user was blocked on task output because it does not consider the away periods (Chapter 2.3.2) that may have occurred between a task being requested and the task running to completion. The actual time blocked on a task's output, $V_a^{\text{resp}\prime}$, is

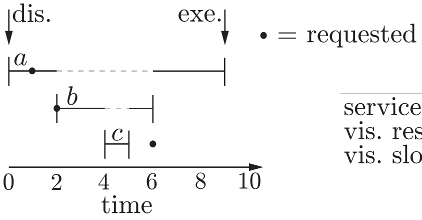
$$V_a^{\text{resp}\prime} \overset{\text{def}}{=} V_a^{\text{resp}} - \text{ any contemporaneous away periods of the submitter.}$$

I discuss the difficulty in knowing away periods accurately in Chapter 5.7. The results and success of this thesis are not contingent upon obtaining and using away period information. For all scheduling and evaluation purposes, the definition of V_a^{resp} is used.

In addition to visible response time, I also study mean *visible slowdown.* The visible slowdown of one task is

$$V_a^{\text{slow}} \overset{\text{def}}{=} \frac{V_a^{\text{resp}}}{S_a}.$$

Note the differences between these two metrics and the non-speculative scheduling's response time and slowdown from Chapter 4.3. A task's visible



	a	b	c
service time	5	3	1
vis. resp. time	8	4	0
vis. slowdown	1.6	1.3	0

Figure 5.6: How when a task is disclosed, requested, and executed, along with a task's service time, determines its visible response time and visible slowdown in the context of batchactive scheduling. (Compare to Figure 4.5.) Three tasks are shown. Requests are indicted by dots. If a request — which indicates task output need — happens after a task executes, as in task c, then its deadline was met and its visible response time is 0. In this particular scheduling policy, shorter tasks preempt (indicated by a dotted line) larger tasks. (This is not a two-tiered batchactive policy as defined in Chapter 5.5.1 because the disclosed task c executes while other requested tasks exist.)

response time may be less than its service time. Further, visible slowdown may be less than 1. These are because if the task had been disclosed, it may have been chosen to run before being needed. Or, if the task had been requested before being needed under a non-speculative scheduler, it also may have been chosen to run before being needed. (Assuming $S_a > 0$, as is only reasonable, both visible metrics are non-negative and finite.) Visible response time and visible slowdown are depicted in Figure 5.6.

The throughput metric is also refined. *Visible throughput* represents only the number of needed tasks which completed (whether under a batchactive or non-speculative scheduler). That is, only the subset of executed tasks that are known or are eventually known to be needed by the user contribute to visible throughput. (Visible throughput could be lower than traditional throughput if the scheduler runs the wrong speculative tasks.) Users using a speculative scheduler disclose speculative tasks and request needed tasks. Here, only the requested tasks that execute to completion contribute to visible throughput, i.e., the *finished* tasks.

Another metric is the *variance of visible response time.* Recall that users desire consistency in the time that they are blocked on task outputs (Chapter 4.4.1). The results of this thesis (Chapter 6.2) tabulate this metric to ensure that it does not get worse with speculative schedulers.

The *number of deadlines met* over some time is the number of times that the visible response time of a task was 0;[3] i.e., the number of times that

[3] 'Deadline' is used in analogy to its real-time scheduling meaning (Chapter 5.6.5).

an eventually needed task executed before being needed. This immediate turnaround time is impossible unless a user submits a task — a request with a non-speculative scheduler or a disclosure with a speculative scheduler — before needing it.

Consider the total requested resources used by a user. The variance of this value across users is called the *variance of user requested resource usage*. This metric reflects the extent to which users use different amounts of resources for tasks that began or eventually were determined to be needed. The lower this variance, the closer users were to using the same amounts of requested resources over time. This metric differs from the variance of user resource usage introduced in the non-speculative environment (Chapter 4.3) because it ignores disclosed tasks that are never requested.

Mean scaled billed resources, with the same definition as in the non-speculative scheduling environment (Chapter 4.3), is an important metric in the speculative scheduling environment. It is always optimal (1) under the batchactive pricing mechanism (Chapter 5.1): no resources are billed for disclosed tasks that were never requested. The batchactive pricing mechanism, by supporting uncharged task disclosure, removes the need for users to make response time / cost tradeoffs of each task submission. Any task that is not known to be needed (i.e., any speculative task) is not initially requested, but disclosed. But under the non-speculative scheduling pricing mechanism (Chapter 4.2), users who request speculative tasks that they later find they will not need are charged for the resources these tasks consume.

The batchactive environment tracks three load-related metrics. *Load* is the same as in the non-speculative environment: the fraction of time that the server was doing work during some time period, irrespective of whether the work was requested or disclosed.

The remaining two load variants concern server revenue. *Requested load* tracks the fraction of time that a server was doing work that was eventually requested during some time period. This metric is needed to compare the revenue of a server that sells resource time under the batchactive pricing mechanism (Chapter 5.1) against the pricing mechanism of charging for all resource use (Chapter 4.2). Another way to understand the implications of the batchactive pricing mechanism is with *uncharged load,* which is simply load minus requested load.

I summarize these metrics in Table 5.1. Two metrics introduced in the context of non-speculative scheduling (Chapter 4.3) were not covered here because scheduling goals based on them were later deemed unimportant or outside of my scope (Chapter 4.4). These were the variance of instantaneous slowdown (degree of equal- or fair-share) and the maximum over

metric	description
mean visible response time	average blocked time
mean visible slowdown	average blocked time scaled by task size
visible task throughput	number of needed tasks that completed
variance of visible response time	how visible response times differ
number of deadlines met	number of tasks executed before being needed
variance of user requested resource usage	degree of per-user equal requested resource usage
mean scaled billed resources	average per-user billed over needed resources
load	fraction of server busy time
requested load	fraction of time taken by requested tasks
uncharged load	load minus requested load
decision count	number of scheduling decisions

Table 5.1: Revised scheduling metrics when allowing for speculative scheduling. This list is a generalization over the non-speculative scheduling metrics listed in Table 4.1. Visible response time and visible slowdown replace response time and slowdown. Visible task throughput refers to only executed tasks that were also needed. The variance of visible response time replaces the mean variance of user response time. The number of deadlines met is a new metric. The variance of user requested resource usage ignores resources used by disclosed but never requested tasks. Scaled billed resources is unchanged. Load is refined based on whether tasks were requested. Finally, decision count is unchanged.

mean slowdown (degree of starvation). All other non-speculative scheduling metrics have analogous speculative scheduling counterparts.

5.3 Batchactive scheduling goals

There are batchactive user goals and batchactive resource provider goals. The scheduling metrics that I refer to in defining these goals were formalized in Chapter 5.2 and summarized in Table 5.1. Goals with analogs in the non-speculative environment (Chapter 4.4) are covered quickly while differences are emphasized.

5.3.1 Batchactive user goals

Users wish the minimum time between needing and receiving task output. This time may be scaled by task size. That is, the scheduler should *minimize mean visible response time* and *minimize mean visible slowdown.* These are speculation-aware versions of minimizing mean response and minimizing mean slowdown from Chapter 4.4.1.

Users also wish to reduce the amount they pay for unneeded resources. As in Chapter 4.4.1, minimizing this cost is not a scheduling problem, and thus is not considered a scheduling goal. The employed pricing mechanism impacts what users pay for unneeded tasks, which is reflected by the mean scaled billed resource metric. Under the batchactive pricing mechanism, this value is always optimal (1); the results (Chapter 6.2) of this thesis compare the extent to which non-speculative scheduling produces a mean scaled billed resources over 1.

When resources are not directly charged (such as in a communal cost-center), the goal is to *minimize the variance of user requested resource usage* (while remaining work-conserving), because doing so prevents resource abuse, as in the analogous non-speculative scheduling goal (Chapter 4.4.1).

5.3.2 Batchactive resource provider's goals

According to the batchactive pricing mechanism (Chapter 5.1.2), the resource provider's revenue is proportional to the amount of resources requested during some time period.

Thus, from the resource provider's perspective, a scheduler should *maximize requested load,* the amount of time that resources are busy running tasks that are or will be requested. (This is analogous to the non-speculative goal of maximizing load in Chapter 4.4.2, since there, every resource usage is billed.) When a resource is idle, or is running a disclosed task that will not be requested, the provider loses potential revenue. To focus on this lost revenue (instead of conflating unneeded speculation with idle time), a subgoal is to *minimize uncharged load.*

The relationship between requested load and visible task throughput is depicted in Figure 5.7. Save for the effect of task cancelation, requested load and visible task throughput are interchangeable under batchactive scheduling in proportion by the mean service time of requested tasks. These are not interchangeable under non-speculative scheduling: visible task throughput ignores unneeded tasks speculatively requested.

Requested load can be increased by returning output quickly to users so that they will submit new work quickly. Uncharged load can be decreased by executing tasks that have been, or are more likely to be, requested.

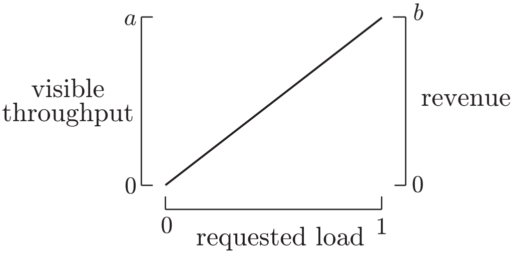


Figure 5.7: How requested load affects visible throughput and revenue under batchactive scheduling. (Figure 4.6 is the analogous sketch for non-speculative scheduling.) As requested load varies, visible throughput and revenue vary proportionally. Visible throughput is scaled by the mean task service time of requested tasks and revenue is scaled by the mean task service time of requested tasks and the price per unit resource. Shown at maximum requested load (1) is maximum visible throughput (a) and maximum server revenue (b). This sketch does not depict the effect of task cancelation: tasks which may have taken some requested load before being canceled should not increase visible throughput.

with respect to	goal
user	minimize mean visible response time
user	minimize mean visible slowdown
user	minimize the var. of user requested resource usage
resource provider	maximize requested load
resource provider	minimize uncharged load

Table 5.2: Speculative scheduling goals. (Compare to the non-speculative scheduling goals of Table 4.2.)

5.3.3 Summary of batchactive scheduling goals

I discussed user goals and resource provider goals separately for speculative scheduling. They were built on the metrics listed in Table 5.1 and are summarized in Table 5.2.

Although, as mentioned in Chapter 4.4.3, minimizing mean response time and mean slowdown sometimes conflict, the best algorithm for minimizing mean visible response time *among the policies discussed* (Chapter 5.5.1) best minimizes mean visible slowdown (Chapter 6.2). To prevent users from dominating resources, where users should be cooperating with shared resources but where such cooperation cannot be assumed, minimizing the variance of user requested resource usage trumps meeting time-based metrics. Maximizing requested load maximizes resource provider revenue, and minimizing uncharged load contributes toward this.

5.4 General batchactive policies

A batchactive scheduling policy uses knowledge that some tasks are speculative. There are many such policies. Policies may require information easy to obtain or uncertain information that must be predicted. There is a qualitative tradeoff between performance and information required, although results suggest that the bulk of the performance gains are possible with little information (Chapter 6.2.9). This section describes the intuition behind policies which put much information to use. Presenting these ambitious policies helps to understand the behavior of the simpler, more easily implementable policies (Chapter 5.5) that this thesis focuses on.

Toward meeting the goals listed in Table 5.2 with an online scheduler (Chapter 4.5), a batchactive scheduling policy may have the following information on candidate tasks ($A^r(t) \cup A^{\bar{d}}(t)$) available on which to base decisions:

– task service time;

– task deadline, the time that a speculative task will be requested, if the task turns out to be needed;

– the probability that a speculative task will be requested.

For requested tasks under a speculative scheduler, the deadline and request time are the same and the probability is moot (i.e., 1). For disclosed tasks, the deadline is some time after disclosure, and the probability is anything between no chance and absolute certainty.

In batchactive scheduling, two separate scheduling worlds converge: response time-based and deadline-based. The response time-based literature is deadline-agnostic, while the deadline-based literature is concerned with whether deadlines are satisfiable. Between these are soft-real-time schedulers which aim to maximize utility, a function of when tasks run [Jensen et al., 1985]. A task's utility is optimal if it runs before its deadline, after which utility decreases.[4] It is unclear how to define utility (and dynamically redefine utility as conditions change) based on uncertain deadlines and task cancelation. According to a scheduling theorist, scheduling speculative tasks to minimize time-based metrics does not have precedent in the literature [Harchol-Balter, 2003a].

It may take a long time for a user to request or cancel a disclosed task. A user may neglect to cancel a known-unneeded task (out of forgetfulness,

[4] More comparisons against real-time scheduling are in Chapter 5.6.5.

sloth, or apathy), leading to a congestion of speculative tasks. One solution is to consider, for scheduling purposes, disclosed tasks that have not been requested for two or three factors of time longer than normal to be canceled. Another solution is to use task request history to more intelligently select among disclosed tasks, as in the implemented HRP and HRR disclosed queue subpolicies described in Chapter 5.5.1.

For each scheduling goal (Table 5.2), I describe policies toward meeting them. Most of these policies are novel and most rely on predictions of deadlines or probabilities of request. Just as it was not certain that task size would be known in the non-speculative scheduling environment (Chapter 4.7), deadline and probability also might not be known. I discuss how these quantities could be predicted in Chapter 5.7. The following major section (Chapter 5.5) describes reduced-information, simpler, more easily deployable policies that were studied in simulation (Chapter 6.2).

5.4.1 Concerning mean visible response time and mean visible slowdown

It has been suggested [Feitelson et al., 1997] that response time (rather than visible response time) has been the primary scheduling metric [Conway et al., 1967, ch. 8] partly because scheduling analysis becomes harder when tasks are needed after disclosure. Disclosure adds one or more of the following difficult considerations: multiple submitted tasks per user, only some of which the user is waiting for, task cancelations, and away periods. These difficulties have confounded theory researchers — I am unaware of results concerning minimizing mean visible response time or visible slowdown — and, although I present results in simulation that include these considerations (Chapter 6.2), I have not provided analysis. (However, I apply operational laws to a non-speculative simulation run to verify aspects of the simulator which have well-known analytic analogues in Chapter 6.1.4.)

For mean response time — in which there is no disclosure and response time is tabulated from request to completion, irrespective of when a user needed a task — SRPT, which only requires task size, is optimal (Chapter 4.5.1). SRPT is also best among the listed non-speculative scheduling policies for mean slowdown (Chapter 4.5.2).

In the context of speculative scheduling, task deadline and probability of a task being needed may also be known. This information can be used to preferentially schedule probable, soon to be needed tasks and delay the execution of less likely to be needed tasks. Deferring such work can prevent running tasks that will later be canceled. It is unclear among size, deadline, and probability which matters most. The intuition of a scheduling theorist is that probability and size are more important than deadline [Harchol-Balter, 2003a]. The items in the following non-comprehensive list place different emphasis on these inputs (all save EDF are novel):

- *Least probability-scaled remaining time* (LPRT).

  Let S_a^{left} denote the remaining service time before completion of task a. This policy selects the task with the lowest $S_a^{\text{left}} \times (1-p)^i$ where p is the probability of the task being requested, and i is some constant determined through experiment. This policy uses size and probability estimates.

- *Least probability-scaled remaining time with postponement* (LPRTP).

  This policy is same as LPRT except that if the nominated task will not be requested for a sufficiently long time (which may be determined through experiment), the task is postponed and the next runner-up is considered. This policy uses size, probability, and rough deadline estimates.

- *Earliest deadline first* (EDF) [Stankovic et al., 1995].

  This policy selects the task whose deadline is soonest. This policy uses deadline estimates.

- *Earliest scaled deadline first* (ESDF).

  This policy is a new variation on EDF which divides deadlines by probabilities and selects the task with the soonest scaled deadline. This policy uses probability and deadline estimates.

EDF and ESDF are difficult to implement because of the difficulty in knowing deadlines (Chapters 5.6.5 and 5.7).

If away periods (Chapter 2.3.2) are also known, the following refinement can apply: Consider a requested task. If the requesting user enters an away period, the task is temporarily considered, for the duration of the user's away period, to have a deadline of when the away period will finish.

The algorithmic complexity of a scheduling algorithm determines runtime when there is a sufficiently large number of tasks. The simplest approach to making a scheduling decision would be to search all tasks organized in an unordered linked list for the appropriate task (such as the task with the earliest deadline, for the EDF policy), resulting in worst-case $O(n)$ performance where n is the number of tasks. This is not satisfying when there exists a large number of tasks as expected when users deeply speculate.

A better approach is a priority queue constructed from a heap, which is common for a wide range of scheduling policies [Cormen et al., 1990]. Extracting the best task and inserting a new task are worst-case $O(\lg n)$ operations using heaps. Each of the above algorithms can use priority queues. However, in the uncommon (relative to other scheduling operations) case that the information that a queue is sorted on (like deadline) change (because better information becomes available), then the heap property needs to be reinstated, a potentially expensive worst-case $O(n \lg n)$ operation. Selecting the best task in the LPRTP policy is likely to be more expensive when many tasks are postponed; at worst, all tasks are postponed leading to a worst-case $O(n)$ performance. One possible fix is to keep all tasks with distant deadlines in another structure and incorporate them in the heap only after some time has passed.

5.4.2 Concerning the variance of user requested resource usage

Recall that user-FB (Chapter 4.5.3) optimally minimizes the variance of user resource usage while remaining work conserving. A small modification, which looks at only the total amount of *requested* resources consumed by a user and which I call user-requested-FB, optimally minimizes the variance of user requested resource usage. When a scheduling decision is to be made, user-requested-FB selects the task from the user that has used the fewest requested resources. If a user has more than one task queued, tasks are taken in FCFS order from that user on the assumption, met by the sequential tasks application type (Chapter 2.2.2) and any-time task sets (Chapter 2.1), that the user will need the tasks according to submission order.

Because running a disclosed task does not affect requested resources, a complete scheduling policy utilizing user-requested-FB must prioritize not only among requested tasks, but also among disclosed tasks. Such two-tiered scheduling is introduced in the next major section, Chapter 5.5.

The policy user-requested-FB can be implemented with a priority queue and thus all operations take $O(\lg n)$ time. [Cormen et al., 1990]

5.4.3 Concerning requested load

If people behave similar to an open system, then requested load can be increased by adding more users. If people behave similar to a closed system, then, in addition to adding more users, requested load can be increased by a policy that also improves visible response time. Those policies, such as favoring small tasks likely to be requested, were covered above in Chapter 5.4.1.

The arguments for how more users and scheduling policy affect load were made in the context of non-speculative scheduling (Chapter 4.5.4). In my target application domain (Chapter 2.1), how people use computing systems combines aspects of an open system (users arriving and departing) and a closed system (users needing and thinking about the output of one task before the next task).

5.4.4 Summary of general batchactive policies

A batchactive scheduling policy uses knowledge that some tasks are not known to be needed at submission time; that some tasks are speculative.

It is unknown how to optimally minimize mean response time or minimizing mean slowdown in the speculative scheduling environment. Intuitively, the more information put to use, such as estimates of task deadlines and the probability of a task being needed, the better these goals can be achieved; e.g., an SRPT variant which scales remaining execution time by 1 minus the probability of request and postpones candidates with late deadlines (LPRTP). More practical approaches toward minimizing visible response time and visible slowdown are introduced in the next major section (Chapter 5.5). Approaches which aid these time-based goals also often increase requested load, the primary resource provider's goal.

When resources are not directly charged and users may abuse resources, the goal of minimizing the variance of user requested resource usage trumps time-based goals and the policy should be user-requested-FB.

Using priority queues, the algorithmic complexity of all scheduling operations for all the policies introduced in this section is $O(\lg n)$ in the worst case.

5.5 Implemented batchactive policies

By not requiring difficult to obtain information, the batchactive policies implemented in this thesis are more easily deployable than some of those discussed earlier (Chapter 5.4). I later demonstrate substantial performance improvements (Chapter 6.2) even when the minimum information (i.e., whether or not a task is speculative) is available.

I begin by defining the two-tiered nature of the implemented schedulers. Then I describe policies that I designed and implemented toward meeting specific scheduling goals. I also show that the two-tiered approach, which always favors known-needed tasks, is not always correct. Finally, I introduce, to later gain insight of how worse than optimal the advocated batchactive

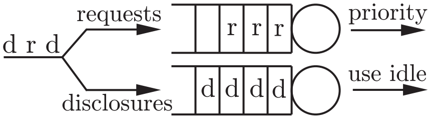


Figure 5.8: The implemented batchactive policies segregate requested and disclosed tasks into different queues in which the requested queue has priority.

schedulers are, an impractical two-tiered scheduler which requires impossible to obtain information.

5.5.1 Two-tiered scheduling

The implemented batchactive scheduling solutions, under the assumption that requested work is more important than speculative work, share the property that requested tasks, $A^r(t)$, have priority. That is, if an idle processor and a pending requested task exist, this task runs before any pending disclosed tasks, $A^{\bar{d}}(t)$, as illustrated in Figure 5.8. Most of the time, this two-tiered approach — having independent queues for requested and disclosed work — is the right choice. (Chapter 5.5.2 offers a counterexample.) Further, those deployed schedulers with baroque requirements (debug queues, administrator queues, 'special' queues, etc., described in Chapter 4.6.1) can be extended with little difficulty to have a disclosed task queue to support speculative tasks.

The overall policy may employ distinct subpolicies for each queue. I notate two-tiered batchactive schedulers as *requested task subpolicy* × *disclosed task subpolicy*. For example, SRPT × FCFS indicates a requested queue subpolicy of SRPT and a disclosed queue subpolicy of FCFS.

Figure 5.9 illustrates two-tiered batchactive scheduling by showing the lengths of such a policy's requested and disclosed queues. The length of the requested queue increases when users request tasks and decreases when these tasks execute or are canceled. The length of the disclosed queue increases when users disclose tasks and decreases when these tasks execute, are requested, or are canceled. Moreover, disclosed tasks only run when there are no requested tasks available to run.

What remains is to determine which subpolicies to employ for the requested and disclosed queues.

Once a task is requested (whether it began that way, or was later promoted from being disclosed), the task looks no different than one in a non-speculative scheduling environment. Thus, a well-studied non-speculative

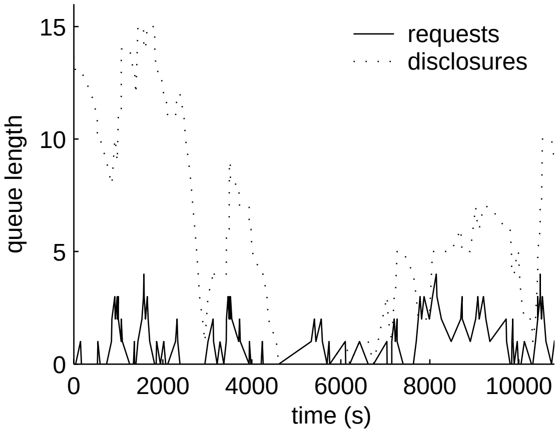


Figure 5.9: Queue lengths of a two-tiered batchactive scheduler over a three-hour simulation of SRPT × FCFS. Running tasks count as being in queue.

policy should be used to schedule requested tasks; the choice of policy based on the arguments listed in the context of non-speculative scheduling (Chapter 4.5) and summarized here.

Recall that when resources are directly charged, the principal goal is to minimize time-based metrics (which also works toward maximizing requested load). Once a task is requested, response time accrues and the best algorithm is SRPT (shortest-remaining-processing-time). Existing systems which do not have service time estimates employ either FCFS (first-come-first-serve) or FB (foreground-background), even through FB would do better in light of task size distributions with decreasing failure rates (Chapter 4.5.1).[5] When resources are not directly charged, the principal goal is to prevent resource abuse. The third and final subpolicy for requested tasks that I study is user-requested-FB, which minimizes the variance of user requested resource usage. SRPT and user-requested-FB can be implemented using priority queues to achieve an algorithmic complexity for all scheduling operations of $O(\lg n)$ [Cormen et al., 1990] in the worst case. FCFS can be implemented with a doubly-linked list for an algorithmic complexity for all scheduling operations of $\Theta(1)$.

There is little theory regarding the proper scheduling of disclosed, speculative tasks with respect to minimizing time-based metrics (and indirectly maximizing requested load). I examine several subpolicies for speculative

[5] The resource provider's reasons for sometimes choosing FCFS over a policy which might do better for minimizing mean response time were described in Chapter 4.6.3.

tasks with respect to the users' time-based goals (which contribute to meeting the resource provider's revenue-based goals for a profit-center). Since a disclosed task subpolicy cannot affect requested resource usage, there are no disclosed task policies for the goal of minimizing the variance of user requested resource usage.

Because of its optimality with respect to minimizing response time, SRPT is initially an obvious choice for the disclosed queue. However, deadlines, which are important for effectively scheduling disclosed tasks, are not considered by SRPT. FCFS is another choice. The motivation for FCFS is to quickly run tasks that will be requested first under the assumption, met by the sequential tasks application type (Chapter 2.2.2) and any-time task sets (Chapter 2.1), that users will request speculative tasks in the order in which they were disclosed. Thus, applying FCFS to all disclosed tasks is a probably a good estimate of request order from one user, and a reasonable estimate across users.

Two novel subpolicies for the disclosed queue leverage the unique features of speculative tasks. The first is called highest-request-probability (HRP) and it favors users who have historically been better speculators; i.e., users who have more often requested tasks that they disclosed. Within a user, tasks are selected according to the order in which they were disclosed. Ways in which to track the likelihood of a user to request a disclosed task are covered in Chapter 5.7. The implemented HRP policy looks at all disclosed tasks from one user and calculates the percentage of them which were eventually requested and uses this percentage as the request probability.

The motivation for HRP is that by avoiding executing speculative tasks that will not be eventually requested, visible response time, visible slowdown, and requested load should improve. HRP also resists a particular type of (intentional or not) denial-of-service on the disclosed queue described in Chapter 5.8. The downside of HRP is that it may cause users to hesitate in disclosing work: if the user feels it is unlikely that he or she will request disclosed tasks, then the user may elect to not disclose deeply to avoid having future speculative tasks deferred.

The second policy is called highest-requested-resources (HRR) and it favors users who have accrued the most requested resources; i.e., users who have paid the most in an attempt to improve time-based metrics and requested load. Within a user, tasks are selected in disclosure order. Like HRP, HRR avoids a denial-of-service on the disclosed queue. Unlike HRP, users may disclose freely. The intent is for HRP and HRR to reward good science: the more carefully someone specifies future work, the better the scheduler performs for that person.

Unexplored variants of HRP and HRR would decay probabilities and requested resource usage over time to support users who change their behaviors. Filters for performing this, balancing stability and agility in the face of changing behavior, were described in Chapter 4.7.

If away periods (Chapter 2.3.2) are also known, the following refinement can apply: Consider a requested task. If the user who requested the task enters an away period before the task executes, then the task is considered disclosed for that duration; i.e., temporarily moved from the requested queue to the disclosed queue.

HRP and HRR can be implemented with priority queues, leading to worst-case $O(\lg n)$ complexity for all scheduling operations [Cormen et al., 1990]. The complexity of the whole policy (combining both subpolicies) is the complexity of the worst subpolicy, which is $O(\lg n)$ in the worst case. Even when many users and tasks are present, execution overhead in my unoptimized implementations of two-tiered batchactive schedulers, which use linked lists instead of heaps, is insignificant (Chapter 7.1.4).

The FreeBSD operating system [FreeBSD, 2004] employs an idle queue that is serviced only when the regular queue is empty.[6] I discuss problems in using this idle queue to implement two-tiered batchactive scheduling in Chapter 5.6.2. The Condor clustering system [Condor, 2003] enables users to prioritize their own tasks. I discuss this interface further when showing how it may be extended to deploy batchactive scheduling (Chapter 7.2.3).

5.5.2 Reasonable, not optimal

The two-tiered scheduling policies are based on the belief that demand work is more important than speculative work. This section shows that favoring requested tasks is not always correct for meeting time-based goals. Depending on deadlines and request probabilities, it is sometimes better for the scheduler to run the speculative tasks first. In short, this can occur when there are small speculative tasks which are likely to be requested competing against a few large requested tasks.

Consider a task A of size 10 requested at time 0 and a task B of size 1 also disclosed at time 0. Task B has a 0.99 chance of being requested at time 1 and a 0.01 chance of never being requested.

The first policy runs requested tasks via SRPT. If task B is requested at time 1, the following occurs: Task A will run from time 0 to time 1. Then task B will run from time 1 (when it is requested) to time 2 (completion).

[6] Tasks may be placed in this queue using the `idprio` command.

Finally, task A will run from time 2 to time 11 (completion). The visible response time for task A is $11 - 0 = 11$ and for task B is $2 - 1 = 1$, resulting in a mean visible response time of $(11 + 1)/2 = 6$. If instead task B is never requested, the mean visible response time is 10 (simply the visible response time of task A running by itself from time 0 to time 10). Factoring in the probability of task B being requested, the expected mean would be $6 \times 0.99 + 10 \times 0.01 = 6.01$.

The competing policy runs speculative tasks first. If task B is requested, the following occurs: Task B will run from time 0 to time 1 (completion). Then task A will run from time 1 to time 11 (completion) resulting in a visible response time of $(0+11)/2 = 5.5$ (the 0 represents task B completing before being requested). If instead task B is never requested, the mean visible response time is 11. Factoring in the probability of task B being requested, the expected mean would be $5.5 \times 0.99 + 11 \times 0.01 = 5.555$. Recall that for the first policy the expected mean was 6.01. Since $5.555 < 6.01$, for this scenario, it is better to favor speculative tasks.

Thus, the pervasive claim (stated by Eggert [2004] and DeGroot [1990] and employed in systems such as Optimistic Make [Bubenik and Zwaenepoel, 1989]) that a scheduling policy should choose requested tasks before disclosed but not (yet) requested tasks is false.

Other cases in which disclosed tasks should run first involve locking dependencies among tasks and resource affinities (processor, memory, storage,[7] and network overheads in task preemption). Neither of these are concerns for the communication patterns and granularity of the application scenarios

[7] One case from Patterson's work on prefetching data from disks [Patterson III, 1997] (Chapter 2.5.3) is illustrative [Patterson, 2004]. One tested application was a link stage in the building of an executable. The linking operation desires data blocks in a non-linear order. The linker, `gnuld`, was modified to issue whole-file prefetches (later this was changed to block-level prefetches). Three cases of a link of a small number of files (making inter-file access time critical) were tested: prefetching off, prefetching on with priority given to demand accesses, and prefetching on with priority given to prefetch accesses. With prefetching off there was a fair number of seeks and the performance was medium. With prefetching on and priority given to demand accesses, performance was pathologically worse. This case caused the disk arm to thrash between large offsets. (The main latency in disk accesses are caused by seeks [Ruemmler and Wilkes, 1994].) The best performance resulted with prefetching on and priority given to prefetches because entire files would be read sequentially. In general, it is not known how to strike a balance between I/O prefetching and demand fetching. Perhaps a system should prefetch only if it is efficient and pays off, considerations that could be adapted to dynamically. It may help to calculate whether a prefetch would complete before some predicted disk idle period before issuing the prefetch [Golding et al., 1995]. In my environment of processor-bound task speculation, the granularity of tasks (Chapter 2.2) is such that the cost to preempt is insignificant; switching from a speculative to demand task does not effect performance.

under consideration (Chapter 2.2).

In spite of the non-optimality of two-tiered batchactive scheduling, significant improvements on common practice are achieved in this thesis. For example, FCFS × FCFS performs at least twice as well for about 20% of the simulated scenarios for mean visible response time (Chapter 6.2.4). The small number of scenarios in which batchactive policies do worse are partially explained by the counterexample in this section.

5.5.3 Impractical policy

I also study an impractical two-tiered scheduler to gain insight on how much better performance can be with impossible-to-obtain information.

I implemented a disclosed queue subpolicy called request-first-come-first-serve (RFCFS) which works like FCFS except that disclosed tasks that will never be requested are never run. That is, the uncharged load of a batchactive policy using RFCFS is zero.

An omniscient scheduler which also knew task deadlines could result in an even better policy. However, while the design of my simulator makes it simple to implement a policy (RFCFS) that knows the probability of a task request perfectly, it is difficult to know its deadline perfectly, which is a function of many dynamic considerations elaborated on in Chapter 5.7. Thus, such a scheduler for comparison purposes is not attempted.

5.5.4 Summary of implemented batchactive policies

The studied batchactive schedulers are all two-tiered schedulers that give preference to requested tasks, are easily implemented, and are easily deployable as an additional disclosed queue to existing computer systems. The requested queue subpolicies SRPT, FCFS, and user-requested-FB follow the non-speculative motivation in Chapter 4.6.3. Two disclosed queue subpolicies are novel: HRP, which favors users who speculate less (i.e., users who submit speculative tasks with high 'hit rates'), and HRR, which favors users who have requested more work over time. One disclosed queue subpolicy, RFCFS, is impractical to implement, but is studied to show how unknown information might improve performance beyond the capabilities of practical batchactive schedulers. The disclosed queue subpolicies are listed in Table 5.3.

I make no claims as to the optimality of the batchactive policies. In fact, I showed that in some situations, the two-tiered approach is wrong. Instead, I demonstrate the superiority of several batchactive policies in simulation

policy	description
FCFS	first-come-first-serve
SRPT	shortest-remaining-processing-time first
HRP	user with the highest-request-probability first
HRR	user with the highest-requested-resources first
RFCFS	run only tasks to be requested via FCFS

Table 5.3: Disclosed queue scheduling subpolicies. A complete two-tiered batchactive policy is formed by taking a non-speculative subpolicy from Table 4.3, such as FCFS, and combining it with a disclosed queue subpolicy from this table, such as HRP, to achieve, e.g., FCFS × HRP.

(Chapter 6.2). By using priority queues, the algorithmic complexity of two-tiered batchactive policies for all scheduling operations is $O(\lg n)$ in the worst case and overhead measurements in Chapter 7.1.4 confirm that the policies are tractable.

5.6 Discordant transformation of existing scheduling

Here I examine how to implement batchactive scheduling on existing interfaces with an eye to whether some or all of the performance benefits of batchactive scheduling can be achieved. I determine that different underlying schedulers can emulate batchactive scheduling to different degrees. The emulations that come closest present awkward interfaces to the user. Thus, even with a underlying scheduler capable of emulating batchactive properties closely, I recommend a batchactive interposition layer between the user and system so that the user is presented with the batchactive scheduling interface, which I believe would be easier to use.

Batchactive scheduling introduces an interface which differs from traditional scheduling because (1) speculative tasks are initially *disclosed,* (2) needed tasks are explicitly *requested* at the time of need, and (3) task output is *isolated* until requested. Disclosure enables a scheduler to treat requested and disclosed tasks differently. Requesting (or ‘pulling’ task output) enables more efficient policies based on learning user behavior and enables the system to provide feedback to the user of the visible response times of his or her tasks.[8] Isolation enables the batchactive pricing mechanism. I argue that the batchactive pricing mechanism is beneficial because it encourages users to issue speculative tasks (Chapter 5.1).

[8] This feedback is useful because it shows the user the benefits of batchactive scheduling.

For example, a simple batchactive scheduler, like FCFS × FCFS, derives performance improvements by giving disclosures lower priority. Some batchactive schedulers, like FCFS × HRP, provide additional performance by learning user behavior. They do so by examining when disclosures are eventually (or never) needed via explicit task request (or cancelation) commands. Knowing when a task was needed enables the calculation of visible response time. The batchactive pricing mechanism is enabled by the isolation of task output: if task outputs were not isolated, a user may attempt to read needed output that was speculatively generated without requesting and being charged for the output.

Over time, different kinds of schedulers have been developed to address the needs of different kinds of applications and operating environments. I examine how interfaces of existing schedulers inhibit or permit task discrimination, the learning of user behavior, the reporting of visible response time, and the application of the batchactive pricing mechanism. I map to batchactive scheduling to the following: standard Unix scheduling using the `nice` priority interface or signal delivery, priority-class scheduling (using FreeBSD's idle queue scheduler as a specific example), and the Condor clustering system's scheduling. For each example I discuss separately how task disclosure can be emulated to achieve task discrimination. I also discuss proportional-share and real-time scheduling. A file system approach to achieving the other batchactive properties is discussed last and is applicable to all the underlying schedulers under consideration. Users would be motivated to use the mappings I present by the positive results of this thesis (Chapter 6.2).

5.6.1 Applying Unix scheduling

Unix scheduling [McKusick et al., 1996] targets multiple users time-sharing a single server running quick-response tasks along with long-running tasks. Unix scheduling strives to improve the response time of users on consoles interacting with shells, editors, graphical user interfaces, etc., while ensuring that software builds, scientific computations, etc., do not starve. For the latter kinds of tasks, slowdown may better reflect user expectations. At the same time, the scheduler strives to provision resources 'fairly' (equal-share) among tasks. Tasks blocking on other resources are replaced by other candidates for execution to pipeline the use of multiple resources among multiple tasks. [Valhalia, 1995; Tanenbaum, 1992]

Unix scheduling is found in FreeBSD [FreeBSD, 2004], System V [Goodheart and Cox, 1994], Mach [Black, 1991], among other operating systems. It employs a multi-level feedback queue in which tasks with equal priority resides on the same queue. The scheduler runs tasks round-robin from the highest priority non-empty queue. The scheduler favors small tasks by lowering the priority of tasks as they consume processor time and by preempting tasks before their quanta expire if a higher priority sleeping process wakes up. The scheduler prevents starvation by periodically raising the priority of tasks that have not recently run.

Unix scheduling poorly meets time-based user goals (Table 5.2) because it executes all of a user's speculative tasks at once instead of in the needed order and because speculative and non-speculative tasks across users run at once. Unix scheduling poorly meets the resource-based user goal of minimizing the variance of user requested resource usage because it strives for fairness among tasks, not among users. Because scheduling does not meet time-based user goals well, scheduling will fall short of the resource provider goal of maximizing requested load (Chapter 5.4.3).

There are at least two ways in which one may adapt standard Unix scheduling to run speculative tasks at lower priority: adjusting task priorities using the `nice` interface and stopping (pausing) / resuming tasks using signals.

Unix scheduling exposes an interface to influence scheduling priority called `nice`. A `nice` value is an integer ranging from -20 to 20, in which lower numbers indicate higher priority.[9] Users would submit speculative tasks at a `nice` value of 20 using the `nice` command. If a user needs a task's output and the task has not yet executed, the user would minimize the task's `nice` value. These `nice` value changes would be easier to accomplish through a batchactive translation tool. Given a user's task set, all but the first task can be issued by the tool at the maximum `nice` value. If the task has not executed by the time of need, the tool can reset the task's `nice` value.

This approach does not match two-tiered batchactive scheduling (Chapter 5.5.1). Speculative tasks will timeshare the processor with needed tasks as the internal priorities in the Unix scheduler's multi-level feedback queue are decayed. The achieved service rates of Unix scheduling combined with `nice` priority changes is not standardized or well-defined. For example, it has been shown that it is difficult to construct systems based on `nice` to guarantee a portion of the processor resource. [Hellerstein, 1993; Straathof et al., 1986].

Unix scheduling supports the delivery of signals [McKusick et al., 1996] to

[9] Only the superuser can specify `nice` values less than 0, thus the range is effectively 0 to 20 without a '`setuid` root' executable that could be part of a batchactive toolkit.

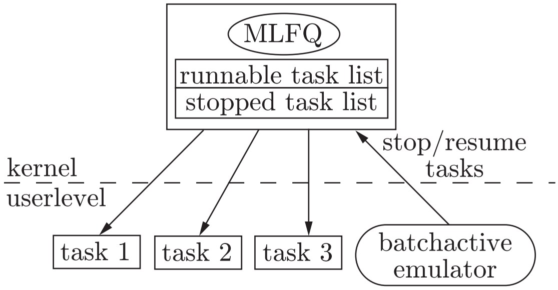


Figure 5.10: Emulating batchactive scheduling on Unix scheduling. Shown is a user-level batchactive scheduler issuing stop / resume signals to the Unix scheduler to control speculative and non-speculative tasks. Only one task will be eligible to run in the kernel's multi-level feedback queue (MLFQ) at any time. This method effectively removes scheduling decisions from Unix.

unconditionally stop and resume the execution of a task (using `SIGSTOP` and `SIGCONT`, respectively). A more direct way to emulate two-tiered batchactive scheduling would be to stop the execution of any speculative tasks when non-speculative tasks are ready to run. This would achieve a variant of FB × FB in which resource usage is decayed. It is overly burdensome for a user to be aware of when non-speculative tasks are ready to run. Thus, a monitoring tool would determine when such signals should be issued. More control, to achieve policies such as FCFS × FCFS, e.g., would be possible by this tool stopping even non-speculative tasks (Figure 5.10). Bypassing the decay-usage prioritization of the kernel's multi-level feedback queue effectively removes scheduling decisions from Unix, but is only possible if scheduling decisions are not frequent, to avoid excessive user-level scheduling and signal delivery overhead.

5.6.2 Applying priority-class scheduling

Two-tiered batchactive scheduling is a restricted case of priority-based queues or classed schedulers [Corbató et al., 1962]. Classed schedulers pick from the highest non-empty class before proceeding to lower classes. Within a class, tasks may run to completion or processor-share, among other policies. Such scheduling is employed to pigeonhole users into different classes of service (i.e., providing 'differential' service) by an administrator, based, perhaps, on advertised prices, service level agreements, importance, or on recent resource usage such as in the multi-level feedback queue of Unix (Chapter 5.6.1). For example, the tasks of a principal investigator might rank higher than a graduate student's, which might rank higher than an undergraduate research assistant's. Such rankings are not difficult to create: psychological evidence suggests that people are good at finding the conditionally monotone relationships necessary in forming them [Dawes, 1979]. In contrast, batchactive scheduling employs only two queues or classes and each user has access to both by choosing to request a needed or disclose a speculative task.

Two-tiered batchactive scheduling can be emulated on a classed scheduler with two classes as follows. A user could place speculative tasks in to the lower priority queue and needed tasks that have not executed into the higher priority queue.

A batchactive translation tool could abstract away queue details. Given a user's task set, the tool would submit all but the first task into the lower-priority class and the first task into the higher-priority class. When a user is blocked on task output and the task has not executed, a command to the tool would move a task from the low priority queue to the high priority queue.

As mentioned in Chapter 5.5.1, the FreeBSD operating system [FreeBSD, 2004] employs an idle queue that is serviced only when the regular queue is empty. FreeBSD exposes an interface to place tasks in an idle queue that only runs when the regular queue is empty using the `idprio` command. Speculative tasks may be placed in the idle queue so that they would not interfere with non-speculative tasks, simulating the two-tiered aspect of two-tiered batchactive scheduling. Again a batchactive translation tool could abstract away the details of executing queue changing commands.

Unfortunately, the underlying policies of the classes might not be modifiable. Typically, the tasks within a class will be serviced via PS (approximated by round-robin with a certain quantum size), FCFS, or FB (also approximated). For FreeBSD, the requested and disclosed queue subpolicies would be the variant of FB which the multi-level feedback queue implements. Source code modification of the scheduler would be necessary to install different policies.

5.6.3 Applying Condor scheduling

The Condor clustering system [Condor, 2003] strives to provide equal resource usage among users. The policy looks at the sum of the resource usage of all the tasks requested by a user and preferentially schedules tasks belonging to users whose tasks have used fewer resources, preventing a user from obtaining more than his 'fair' share by queuing large amounts of work. The amount of resources used by a user's tasks are decayed over time and the inverse ratios of these decayed values across users determines the ratios of resources the users will receive if they queue work, ensuring that tasks belonging to users whose tasks have used more resources do not starve.

Condor enables users to order their own tasks using a `rank` characteristic associated with each submission. However, this characteristic only orders task execution within each user. There is no mechanism to specify that some tasks of one user (the requests) should have a higher priority than some tasks of another user (the disclosures). A proposed solution, involving, in part, submitting tasks from multiple users as if the tasks came from the same user, is described in Chapter 7.2.3.

5.6.4 Applying proportional-share scheduling

A proportional-share scheduler aims to give externally defined fractions of processing time to competing tasks. A user running several processor-bound tasks, such as those found in scientific environments, could control the share of processor time that each task receives, and an administrator could control the relative rates at which different users could use the processor, obtaining load insulation.

Waldspurger et al. introduced a ticket abstraction for flexible resource allocation using novel proportional-share algorithms. In lottery scheduling [Waldspurger and Weihl, 1994], each task holds a number of tickets. The scheduler selects which task to run by picking a ticket from the runnable tasks at random and choosing the task that holds this winning ticket. Stride scheduling [Waldspurger and Weihl, 1995] is a deterministic version of this policy. Cooperating tasks can transfer tickets among each other and 'currencies' enable isolation among users, groups, and subgroups of tasks. Cross-node ticket distribution for enabling proportional-share on a cluster was discussed by Arpaci-Dusseau and Culler [1997]. My extensions to lottery scheduling enabled the coexistence of tasks that demand low dispatch latency and tasks that are processor-bound [Petrou et al., 1999]. Additional proportional-share goals can be achieved with the ticket abstraction [Waldspurger and Weihl, 1996], but they are not relevant to the speculative tasks (Chapter 2.2) under consideration.

With fewer theoretical guarantees, schedulers have been introduced to achieve long-term resource shares among competing users [Larmouth, 1978; Essick, 1990]. In network scheduling, which has the additional challenge of controlling poorly behaving sources which could swamp intermediate routers, deficit round-robin [Shreedhar and Varghese, 1996], built on the 'fair queuing' concepts in Demers et al. [1989], efficiently and with tight bounds proportionally shares network resources.

Proportional-share scheduling is not intended to minimize mean visible response time or minimize mean visible slowdown, the batchactive time-based goals. If each task is given an equal portion, scheduling devolves to PS, and under the common task service time distributions which have decreasing failure rates (Chapter 4.5.1), PS achieves a lower mean response time compared to FCFS but a higher mean response time compared to FB (Chapter 4.5.1). That time-based goals are not achieved prevents the achievement of the resource provider's goal of maximizing requested load (Chapter 5.4.3). Even meeting the resource-based goal of minimizing the variance of user requested resource usage is difficult. Assigning users equal tickets that fund the tickets of the users' tasks does not work because scheduling decisions look at current ticket holdings without knowledge of past resource consumption.

The segregation of speculative and non-speculative tasks to emulate two-tiered batchactive scheduling could be achieved with a proportional-share scheduler by placing each kind of task in a different currency. When a non-speculative task exists, the execution of speculative tasks can be prevented by removing their backing currency. However, the scheduling within each kind of task remains primitive unless tickets are adjusted at every scheduling decision.

5.6.5 Applying real-time scheduling

Real-time scheduling is the catch-all for time-critical demands, providing predictable scheduling with guaranteed or statistically bounded metrics related to whether task-specific deadlines are met usually with resource reservations (resource requirement specifications and admission control) unsuitable to the dynamic application domain of this thesis (Chapter 2.2). Unix scheduling, because it lacks admission control, cannot provide such guarantees, even if it had an interface with which to express deadlines [Valhalia, 1995]. Mercer [1992] is an introduction to real-time scheduling.

The characteristic of a soft-real-time system is that the completion of a task (or an operation within a task) has a value to the user that can be expressed as a function of time [Jensen et al., 1985]. An example application has deadlines every 33 ms by which a video frame must be displayed. Missing these deadlines results in decreased, but non-zero, value, or utility to the user.

The Rialto scheduler [Jones et al., 1997] is an example of a modern real-time scheduler. The SMART scheduler [Nieh and Lam, 1997] is another, and during overload, it notifies applications when their deadlines cannot

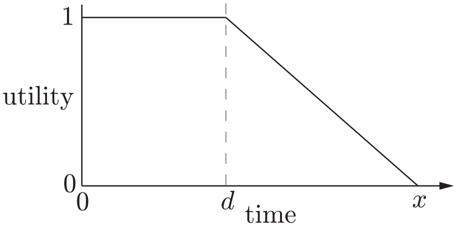


Figure 5.11: The difficulty of mapping utility functions to batchactive scheduling goals. Shown is the utility of a task as a function of when it completes. If it completes before the deadline d, which is determined partially by previous task service time and current think time and away periods, its utility is maximum (1). After this, visible response time accrues and its utility decreases to 0 at time x. Real-time schedulers can use such utility functions to schedule multiple tasks in an attempt to maximize global utility. However, in my environment, deadlines are difficult to predict and tasks may be canceled. Further, it is unclear how utility should degrade (the shape of the curve between times d and x, and the location of x) after the deadline.

be met, allowing adaptive applications to adjust their demand. A simple and well-studied real-time policy is earliest deadline first [Stankovic et al., 1995]. In Chapter 5.4.1 I discussed it and a variant in the context of general batchactive schedulers.

Time-based goals can be attempted by ensuring that tasks finish before their deadlines or quickly after deadlines elapse, thus also meeting the resource provider's goal of maximizing requested load (Chapter 5.4.3). Yet, soft-real-time scheduling is hard to apply because their utility functions are hard to create as depicted in Figure 5.11. The work patterns (Chapter 2.1) of the users of the applications under consideration (Chapter 2.2) are dynamic: deadlines change, think time and away periods are uncertain, tasks are canceled, disclosed tasks may never be requested. While service time, which partially determines deadlines, may be predicted (Chapter 4.7), I argue in Chapter 5.7 that other aspects make it hard to predict deadlines, making real-time scheduling unsuitable even if it could support dynamic deadline changes. If reasonable utility functions can be determined, then scheduling becomes an optimization problem to maximize aggregate utility. Similarly, a utility function mapping resource usage to utility can be computed for the resource-based goal.

5.6.6 Knowing whether a task is desired

I described ways in which different kinds of scheduling interfaces may be extended so that the scheduler favors needed tasks over speculative tasks. In addition, mechanisms are needed to enable smarter scheduling that learns from user behavior, to provide visible response time feedback, and to support the batchactive pricing mechanism (Chapter 5.1).

Consider the HRP disclosed queue subpolicy. Among the studied policies, it provides the best batchactive performance as shown in the results of this thesis. Further, it is robust to a disclosed queue denial-of-service described in Chapter 5.8 that is possible under the batchactive pricing mechanism. HRP requires knowledge of what tasks, out of all disclosed, are eventually needed.

Assume that the underlying scheduler can be modified to support learning subpolicies like HRP.[10] Existing interfaces, unfortunately, do not provide enough information to support such policies. Many task requests and cancelations, which represent whether or not a user desires a task's output, will not be seen by the system. If a needed task executes before the user needs its output, the user will likely directly consume the task output (which is often stored on a distributed file system). If an unneeded task executes before the user knows that its output is unneeded, the user has no motivation to cancel the task.[11] In fact, under HRP, a user does not wish to appear to not need tasks, because HRP favors the users who speculate less. Further, not knowing task request time prevents policies (not simulated in the results of this thesis) which rely on think time estimates, like EDF (Chapter 5.4.1).

If the time of task need is presented to the scheduler via explicit request, then the user can be provided with visible response time statistics, so that the user can appreciate the benefits provided by batchactive scheduling. Visible response time for a task would be the time of task execution minus the time of task request, or 0 if the request happened after execution (Chapter 5.2).

Finally, the traditional interface of not isolating output until needed makes the batchactive pricing mechanism impossible, for without isolating output a user may never request (and pay) for needed output.

[10] Supporting different scheduling, such as FCFS × HRP, would requires source code modification for many systems.

[11] In batchactive scheduling, there is motivation to cancel unneeded tasks. Because task output is isolated, the user does not know if the task has completed. The user will wish to cancel unneeded work so that it would not compete with future tasks that the user might need.

What is required is an emulation of a pull interface for needed tasks and motivation for unneeded tasks to be canceled. The Xgrid clustering system [Xgrid, 2004] is one system that presents a pull interface for task output. Because it is not popular, and because it provides no mechanism for task prioritization needed to emulate two-tiered batchactive scheduling, it is not discussed further. Note that the translation tools described above could support the delivery of task outputs to users, blocking if a needed task is not finished, or returning immediately if so. However, unless this is the only way for a user to receive task output, users might bypass such as tool to directly access the file system to retrieve outputs.

Whether and when a user needs a task's output can be determined outside of an existing scheduler's interface by using file system techniques. One technique relies on file systems that enable hooks to be triggered on certain operations. Another technique relies on monitoring a distributed file system.

DMAPI (data management application programming interface) [The Open Group, 1997] is a file system interface standard that supports the triggering of hooks on specified file system operations. It is commonly used in HSM (hierarchical storage management) systems that adaptively move data based on frequency of use between cheaper / slower and costlier / faster storage layers. An interface to enable interposed layers in the file system that also can trace file system activity was explored by Heidemann and Popek [1994].

NFS [Shepler et al., 2000] monitoring, which may [Blaze, 1992] or may not [Aranya et al., 2004] be passive, is another technique to trace file system operations.

With such techniques, the attempt to read output from a task can be assumed to represent that the user desires the output. The time that the user initiates the read can be assumed to be when the output is desired. These attempted reads (which may yield no data if the task has not completed) would trigger upcalls that supply the statistics necessary for learning policies, calculating visible response time, and charging the user under the batchactive pricing mechanism.

To encourage task cancelation when the user determines that a task is not needed, the user should not be able to find out whether a task has completed. For example, the completion of the task should be hidden by controlling access to administrative commands that display executing commands, such as `ps`. This way, whether or not the task has executed, the user will wish to cancel it so that it would not compete with his or her other tasks. If the task's output consumes accounted storage space, then the user deleting a task's output (which may be an empty file is the task has not executed) before reading it could be considered cancelation.

While useful for the reasons shown, file system-level techniques require a fair bit of mechanism that is not always available.

5.6.7 Summary of the discordant transformation of existing scheduling

In summary, I have shown that existing interfaces do not need to change to get most of the benefits of batchactive scheduling. However, increasingly accurate batchactive emulation requires increasingly arcane techniques and present increasingly awkward interfaces to the user.

The existence of ways to emulate batchactive characteristics simplifies deployment. I recommend that users interact with a batchactive layer, disclosing and requesting, instead of the underlying system, for ease of use.

5.7 Predicting request probability and deadline of speculative tasks

I describe techniques to predict information that some batchactive schedulers require. The techniques for predicting the probability of a speculative task being eventually needed are straightforward. However, it is difficult to predict the deadline of a speculative task (i.e., when it will be requested), suggesting that batchactive policies needing this information are impractical. The prediction of task service time, also needed by some batchactive policies, was covered in Chapter 4.7. Away period prediction is also discussed. Note that batchactive policies needing none of this information, such as FCFS × FCFS, still perform better in many cases than their non-speculative counterparts (Chapter 6.2.4) for the scenarios that I examined. The information that the predictive techniques of this section may provide are intended to aid in obtaining additional performance improvements.

The probability of a disclosed task being needed can be estimated using the same techniques for predicting task service time (Chapter 4.7). There are two types of techniques: predictions based on task characteristics and predictions based on recent user behavior. The former learns relationships between discrete or continuous task parameters and the probability of task request by fitting polynomials. The polynomials can then be evaluated to make probability predictions based on possibly unseen parameters of disclosed tasks that have not yet been requested. The latter looks at the tasks that a user has disclosed and counts which have been eventually requested to determine a ratio or ‘hit-rate’ modeling how well the user tends to speculate. This technique should employ an EWMA or ‘flip-flop’ filter to favor recent behavior.

The deadline of a speculative task is difficult to predict. One can assume that a user will request a needed speculative task some time after he or she has received the output of a previously requested task and has 'thought about' that output; i.e., the time of the previously needed task completing plus the user's think time. Knowing when the previously needed task completed is intractable as it depends on task arrivals, load, scheduling policy, and other dynamic considerations. A simplification, which introduces considerable inaccuracy whose performance implications are unknown, is to assume that the deadline is infinite until the previous task's output is delivered to the user, at which point the deadline becomes the user's think time.

Per-user think time can be predicted by taking the average of the elapsed time between a user receiving task output and requesting the next task's output. This average should be weighed to favor recent information more heavily using the filters discussed in Chapter 4.7.

Another approach is to assume that think time follows a distribution, enabling the system to predict whether the think time associated with one task will exceed that of another task, and thus, to predict whether one deadline will occur before another. Crovella and Bestavros [1995] found that web think times conform to a heavy-tailed distribution. In particular, web client think time is Pareto distributed (Chapter 6.1.3) with an α parameter of 1.5. It is reasonable to assume that think time for non-web browsing tasks is also Pareto distributed.

Away periods (Chapter 2.3.2) are the most ambitious quantities to predict. Deadline predictions should be delayed when a user enters an away period (Chapter 5.4.1). The studied two-tiered policies, which do not make use of deadline predictions, could use away period information to temporarily move tasks from the requested queue to the disclosed queue, favoring users who are not in away periods (Chapter 5.5.1).

Away periods can be explicitly indicated. Users, especially in corporate settings, may maintain calendars or schedules of their activities, indicating, among other responsibilities, lunches, meetings, and presentations. It may be assumed that during such times the user is unavailable to consume task output, i.e., that the user is 'away.' The downside of relying on this information is that calendars, if available at all, are often merely hints as to what a user would do. Further, there is no pressure for a user to comprehensively report his or her away periods, since if the user overreports, his or her tasks suffer.

A scheduler may assume that the user is in an away period if he or she is not in the office. Within an organization, it has been suggested that users wear badges indicating their location [Want et al., 1992]. However, it is difficult and questionable to force people to wear devices that would reveal, e.g., that they were out to lunch when they had claimed they were waiting for their tasks to complete.

Instead, away periods could be estimated by tracking when a user's workstation is idle. Systems that track workstation idleness for remote task execution differ in how they define idleness (e.g., low load, console inactivity for a certain number of minutes, no resident foreign tasks [Douglis and Ousterhout, 1991]). As in predicting think time, away period predictions can be done by analyzing the recent past or by assuming that the periods fit a distribution. Recent workstation idle periods can be averaged and filtered using the techniques in Chapter 4.7. Acharya et al. [1997] found, among other useful statistics, that on average, a workstation that has been idle for five minutes can be expected to be idle for another 40–90 minutes. Other research, however, suggests good predictive power with more complex schemes incorporating more information, such as weekly patterns [Wyckoff et al., 1998; Petrou et al., 1996].

5.8 Preventing resource abuse

Finite computing resources will be overloaded if one is able to run tasks whenever one pleases. As shown by Hardin [1968] using the example of overpopulation, the goal of 'the greatest good for the greatest number' [Bentham, 1823] is impossible in that it tries to maximize two variables, population and happiness, that are in conflict: increasing the population (number of tasks) requires reducing the calories each consumes (increasing visible response time).

The employed pricing mechanism and scheduling policy determine or restrict the ways in which users may intentionally or unintentionally abuse resources.[12] Loosely, resource abuse is the use of resources against the spirit of the system, such as a small number of users consuming an 'unfair' amount of resources. This definition is clarified by the examples of resource abuse below.

Under the batchactive pricing mechanism, either all resource usage is

[12] Unix-like scheduling allows one to create an unbounded number of processes, unfairly swamping the system. Processor-bound tasks receive lower priority, but tasks that interact with users receive higher priority. An old loophole [Tanenbaum, 1992], not present on modern Unix schedulers, was to make processor-bound tasks appear to interact with users (perhaps by having a thread consume otherwise useless user input) to achieve a greater share of the resource.

not directly charged to the user or only requested resources are charged (Chapter 5.1).

An example of abuse is a user requesting instead of disclosing tasks that are speculative. This causes the two-tiered scheduling policies to give those speculative tasks as much priority as requested tasks, which will result in higher mean visible response times; a 'tragedy of the commons.' [Hardin, 1968] When requested resources are charged to the user, this abuse is not possible. If a user attempts to gain scheduling priority by requesting instead of disclosing speculative tasks, then the user will be charged for the resources that those speculative tasks consume.

When resource usage is not directly charged (such as in a communal cost-center), there is no technical solution[13] to the kind of resource abuse in which a user requests tasks that are actually speculative. The best approach is to set the requested queue subpolicy to user-requested-FB to prevent users from consuming more requested resources than other users when other work is available to run (Chapter 5.5.1). This is the same strategy used by non-speculative schedulers to prevent unfair resource usage. Still, if most users tagged speculative work as needed, then the performance of batchactive scheduling would diminish to that of a non-speculative scheduler.

One might assume that users would act honestly in settings where resources are not directly charged and users work toward a shared goal; such as an in-house cluster used by computer animators. However, cluster usage is not directly charged in university settings, yet the users (e.g., graduate students) may or may not collect into larger aggregates with a shared goal; often they compete for resources among one another.

Adam Smith popularized 'the invisible hand' in which an individual who 'intends only his gain' is 'led by an invisible hand to promote [...] the public interest.' [Smith, 1776] Yet experience and analysis have shown the opposite consequence from selfish actions. Consider the following example from Hardin [1968]: in a common pasture, it is in a herdsman interest to add an extra animal, even though doing so contributes to the dissipation of resources because the costs of over-grazing are shared among all herdsmen. In a computing environment, laissez-faire policies result in a user adding more work; it is in one's interest to request work even under contention, when this would decrease responsiveness for all.

Users must be pressured to request only known-needed tasks. One way is to establish social norms. With a scoreboard showing the computational re-

[13] A technical solution may be defined as one that requires a change of scheduling policy only, demanding no change in user morality or intelligence.

sources each person received for requested tasks, abusers can be confronted by peers or privately by IT managers. Such norms have been shown to work well in small groups, but they have not been shown to scale to larger communities [Burger and Gochfeld, 1998]. Also, this appeal to conscience leads to a double-bind: if one does not behave acceptably, he or she is condemned, but if one complies, he or she is secretly condemned for being a simpleton who does not take advantage of the system.

A restrictive and impractical solution is to privatize computing resources so that they are no longer shared. While avoiding resource abuse since each resource is used by only one self-interested entity, this would lead to poor resource utilization. Splitting a resource into portions belonging to individuals, despite the technical and maintenance impracticalities, would result in the inability to support bursts of user activity greater than the portion that he or she controls.

The solution that works is for users to be penalized (charged, taxed) for requested resource usage. Besides the batchactive pricing mechanism (Chapter 5.1.2) which charges a constant amount for requested resources, another possibility suggested by MacKie-Mason and Varian [1995] is for people to be charged to the extent they take computer resources from others; what is called congestion or shadow pricing.

Consider another resource abuse scenario consisting of gaming. Gaming can be defined as the behavior of an individual which increases his utility by using the system contrary to its intended use. It is against the spirit of the system for a user to disclose work he knows he will never request. A user is free to do so in batchactive scheduling, because, under the batchactive pricing mechanism when resources cost, the user is not penalized for needless speculation. The question is whether this behavior increases a user's utility, and if so, whether and how such behavior could be countered using a scheduling mechanism. In this situation, one may define an increase in utility to be a decrease in the user's mean visible response time.

First I determine that this gaming attempt does not result in gaming; a user does not benefit from this behavior. Then I consider the effects of such behavior on the system and on other users, as might be caused by a well-meaning but unintelligent user, a user making a poor attempt at gaming, or by a user wishing to cause harm (viz., deny service). I argue that a batchactive scheduler using HRP or HRR as the disclosed queue subpolicy works to counter this errant or abusive behavior.

There is no advantage for a user to add tasks that will never be requested to any part of his or her task set. For such task sets to benefit a user, they would have to cause tasks that the user will actually request to be preferentially scheduled. Consider a scheduling policy which favored users who have disclosed more work to the system. Under this scheduler, a user can game the system by adding needless speculation. However, no batchactive scheduler in this thesis does this. The closest analogy is HRR, which favors the users who have submitted the most *requested* (charged) work. In traditional, non-speculative scheduling, common policies favor users who have submitted *less* work, such as user-FB. To my knowledge, there is no reasonable policy, deployed or theoretical, that has some advantageous characteristics but that succumbs to this form of gaming. For example, neither FCFS, user-FB, nor SRPT as disclosed queue subpolicies would prefer the needed tasks of users who submit unneeded tasks.

One might ask whether the batchactive pricing mechanism could allow a non-thinking or abusive user to behave in such a way as to lower server revenue or to increase the mean visible response times of others. These are undesirable outcomes that should be countered by better scheduling.

Consider a scenario in which all users except one never disclose work they know they will never request. The remaining user instead has a maximum task set change probability, high tasks per task set, high service time, and high think time. In a batchactive scheduler, unless an appropriate disclosed queue subpolicy is employed, the work from the abusive user will compete with other disclosed work, lessening the benefits of batchactive scheduling until it is on par with traditional scheduling. Without the ability to pipeline disclosed work with think time, the non-abusive users' visible response times will increase (i.e., get worse). Further, users will submit work more slowly to the system, resulting is worse requested load (server revenue).

Thus, a malicious or poorly-speculating user could deny the service of other users' speculative tasks by disclosing deeply but never or rarely requesting. These never-to-be requested tasks would compete with speculative tasks that will more likely be requested. Although its benefits are reduced when this occurs, this behavior would not cause batchactive scheduling to do worse than non-speculative scheduling.

Some disclosed queue subpolicies, such as FCFS, SRPT, and user-FB, cannot prevent these negative consequences. What is needed is a policy that considers the amount of disclosed work that is eventually needed.

A solution to this abuse is for the scheduler to favor the disclosed tasks of historically better speculators or higher payers, which is how the HRP and HRR disclosed queue subpolicy operate, respectively. They both are intended to work in the resource provider's interest, who wishes to profit from running tasks that will be requested, and in the interest of other users, whose speculative tasks would suffer from an artificially large disclosed queue. Using HRP or HRR for the disclosed queue subpolicy removes these negative effects. The abusive user's disclosed tasks will not be run before other user's disclosed tasks. Another solution is to consider, for scheduling purposes and when competing disclosed tasks exist, disclosed tasks from users who typically make requests two or three factors of time longer than normal to be canceled.

Note that the described denial-of-service is not critical because it is not anonymous. An observant administrator will take measures against the abusive user. If such abuse occurs often, the described automatic scheduling methods are appropriate, relieving administrator burden.

Besides the protection of abuse, HRP is a valuable disclosed queue subpolicy because it provides better mean visible response time (across all users) compared to FCFS as the disclosed queue subpolicy, especially when there is a lot of variability in how speculatively users behave, as my results show (Chapter 6.2).

In summary, when requested resources are charged, I am unaware of ways in which users can abuse resources. A denial-of-service under the batchactive pricing mechanism can be avoided with the HRP and HRR disclosed queue subpolicies. When resources are not directly charged, performance may be driven down to the level of a non-speculative scheduler by users who request speculative work.

5.9 Beyond centrally scheduled processing resources

This section discusses how speculative work besides computation on centrally scheduled clusters might be better scheduled, illustrating that the space of speculative scheduling extends beyond the focus of this thesis.

The following are considerations when devising a speculative scheduler: The *pricing mechanism* can either not charge for any resource usage, charge for the resources used, charge over some duration regardless of use, etc. The *granularity* of speculative work may range from small to large; something that can complete in a fraction of a second to hours, affecting the benefit of performing the operation speculatively and the cost that the operation would have on other work (such as context- and TLB-switch overheads). Scheduling decisions may be *centralized* or *decentralized* (laissez-faire), in which there might be background load unknown to or unaccounted by the schedulers. Speculative work may modify state visible to the user and if such work is determined to not be needed *rollbacks* need to restore state, adding complexity and time overhead. More generally, these considerations can form a *cost / benefit* analysis to predict whether speculative work should be done at all (i.e., the 'throttling' notion of DeGroot [1990]).

The application scenarios of this thesis (Chapter 2.2) discussed speculative computations of large enough granularities that plateaus of human perception [Nielsen, 1994, ch. 5] and task switching overheads were irrelevant on architectures with centralized scheduling (Chapter 4.1) employing existing and batchactive pricing mechanisms (Chapters 4.2 and 5.1). No rollbacks were required because all speculative state was isolated ('sandboxed') in an output store for future user retrieval.

I first discuss speculation in the context of web page prefetching and then in the context of task scheduling when the centralized scheduler cannot be modified and does not present an interface that could be leveraged to emulate batchactive scheduling as discussed in Chapter 5.6. These diverse environments can be addressed with a similar feedback-based solution, which is described last.

5.9.1 Web document prefetching

Web document prefetching has the strong potential to improve the experience of those browsing the web, as argued in Chapter 2.2.4. A prefetching agent could construct a task set of prefetch candidates, perhaps by looking at the links on the currently displayed page and recursively for the links on prefetched pages or by heeding server-inserted annotations developed through access log analysis [Padmanabhan and Mogul, 1996]. A prefetch scheduler sitting as a proxy between an unmodified browser and the network could determine how many such prefetch tasks to issue.[14] A person selecting a link is equivalent to requesting task output. The time during which a person reads a web page is equivalent to his or her think time. The scheduler could attempt to balance the visible response time that the person experiences with the fractional increase in network usage caused by prefetching.[15] I assume that no kernel modification or router support exists to understand which accesses are speculative.

Network usage is typically charged by time (such as monthly payments) irrespective of usage. There is no incentive to throttle network speculation to be a good 'netizen.' However, a user performing many simultaneous network transfers would wish that speculative work have less priority than

[14] Whether prefetches would be issued in parallel or sequentially is a function of available bandwidth.

[15] The disk space used to store prefetched data is not considered; given the large size of disks and the small size of web documents, any storage used is usually insignificant.

other flows, meaning that there is a cost to performing ineffective prefetches. Scheduling decisions are decentralized: each user's agent independently decides whether to inject speculative requests into the network, and there exists background network load between web clients and servers unaccounted by the scheduler. Rollbacks are unnecessary because unneeded web accesses are never shown; they are simply discarded.

The granularity of a web page transfer can be small, and thus human perception models should be accounted in a non-linear benefit function. If visible response time is below 100 ms, it is known that the scheduler need not try harder: the user will not notice [Nielsen, 1994, ch. 5]. Other rough boundaries later in time (1, 10, 100 seconds) were found through psychological tests and user studies [Newell, 1990, ch. 3] [Nielsen, 1994, ch. 5]. An application that has a range of acceptable visible response time is called 'elastic.' [Neugebauer, 1999]

In this environment, given a set of prefetch candidates, which, if any, should the scheduler issue to maximize some function of visible response time and fractional increase in network usage?

5.9.2 Decentralized speculative task scheduling

I propose a way for the advantages of batchactive scheduling to be obtained when the underlying cluster scheduler cannot be modified to understand the presence of speculative tasks. That is, the scheduler is opaque and does not present an interface that could be leveraged to emulate batchactive scheduling as discussed in Chapter 5.6.

Each user's machine (console, local workstation) issues speculative tasks independently. To the centralized, unmodified cluster scheduler, such tasks appear indistinguishable from non-speculative tasks. The independent user schedulers, or *batchactive frontends,* seek to minimize visible response time on behalf of their users. Doing so requires reducing interference among non-speculative tasks from the same user. At the same time, the frontends should throttle task speculation to 'play nicely' with respect to other users' tasks. If resources are directly charged, frontends should also seek to minimize charges for unneeded speculation. The frontends may base decisions on the recent visible response time experienced by their users, the queue length of the centralized scheduler, and some per-user function of time v. money.[16]

[16] As a starting point, Professor of Economics Ian Walker proposes [Walker, 2002] that the value of someone's hour, v_h, is

$$v_h \overset{\text{def}}{=} \frac{w_h(1-t)}{C},$$

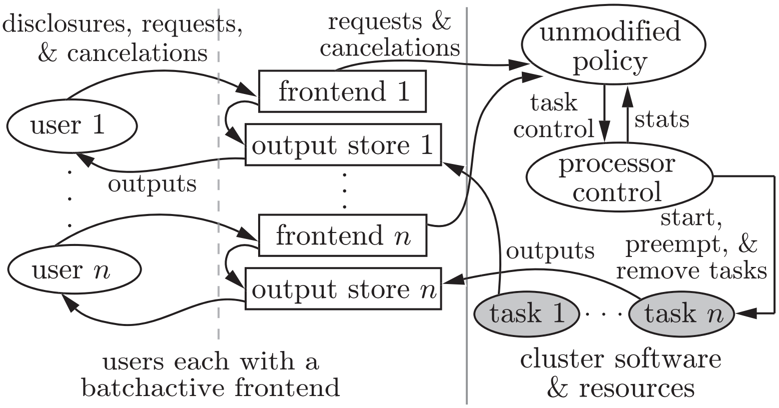


Figure 5.12: Interaction between users, each with a batchactive frontend, and unmodified cluster software and cluster resources. Compare this organization to Figure 5.1, the focus of this thesis, which depicts speculative scheduling when cluster software can be modified.

The choice of the number of speculative tasks to submit that these frontends make is the same burdensome choice that users make when speculating with non-speculative schedulers, one motivator for batchactive scheduling (Chapters 2.4 and 4.8).

Figure 5.12 depicts how users with batchactive frontends interact with the clustering software and resources. Each batchactive frontend independently decides which and when disclosed and requested tasks from a single user pass on to the clustering software's non-speculative policy. The non-speculative policy decides which and when tasks submitted by the frontends run, employing decay-usage or a variant of FCFS (Chapter 4.6.3). This policy communicates with the operating systems running on the cluster resources which handles the details of running tasks on the servers (such as forking processes) and provides task statistics (such as resource usage and load) to the policy. The output from an executed task is delivered to the appropriate frontend's output store and further delivered to the user if and when the task is requested.

Decentralizing speculative decisions at the frontends impacts performance and ease of deployment. The central coordination advocated in this thesis and illustrated in Figure 5.1 should do better at meeting the batchactive goals listed in Table 5.2 than the laissez-faire approach of the indepen-

where w_h is that person's hourly wage, t is the tax rate, and C is the local cost of living index; an advancement over the advice from Benjamin Franklin to a young tradesman in 1748 that 'time is money.'

dent frontends because the single batchactive policy has more information (it receives all user interaction) and has more control (it determines all load).

The frontends are easier to deploy. If a central batchactive policy cannot be installed, then each user who wants some of the benefit of batchactive scheduling can install his or her own batchactive frontend as a library-based batchactive scheduling solution. It is unclear how many users would need to do so for benefits to be significant.

In this environment, given a set of speculative tasks belonging to a user, statistics of the unmodified centralized scheduler such as queue length, and a mapping of time and money for each user, which speculative tasks, if any, should a frontend issue to maximally benefit its user while not overly interfering with the progress of other users' tasks?

5.9.3 Feedback-based approach

Control feedback can support both web document prefetching and decentralized speculative task scheduling. This solution is suited to the decentralized nature of these diverse environments which includes background load unknown or beyond the control of the scheduler. Here, speculation has a price: once a speculative task is injected, it competes against non-speculative tasks both from the same user and other users because the rest of the system cannot treat speculative tasks differently.

A feedback controller for each user could adjust scheduling aggressiveness (whether or not to inject speculative tasks) toward finding the sweet spot maximizing net benefit. Inputs to the controller, such as the recent visible response time of a user or recent server load can be estimated with an EWMA or flip-flop filter (Chapter 4.7). The desired feedback between speculative task output production and needed output consumption is illustrated in Figure 5.13. Interestingly, based on how well a user's visible response time is predicted to be and based on speculation cost, a batchactive frontend may elect to not issue a speculative task (akin to the 'prefetch horizon' in disk prefetching from the work of Patterson et al. [1995]), which may result in the system temporarily becoming non-work-conserving.

Control theory has been applied by systems researchers. Parekh et al. [2001] show how an autoregressive, moving average (ARMA) controller can maintain a given queue length for a Lotus Notes groupware server by selectively rejecting new requests. Abeni et al. [2000] present a controller to find the best processor allocation for an MPEG decoder. Steere et al. [1999] present a controller that adjusts the portion and period of processor time allocated to applications based on the progress that they make. In building

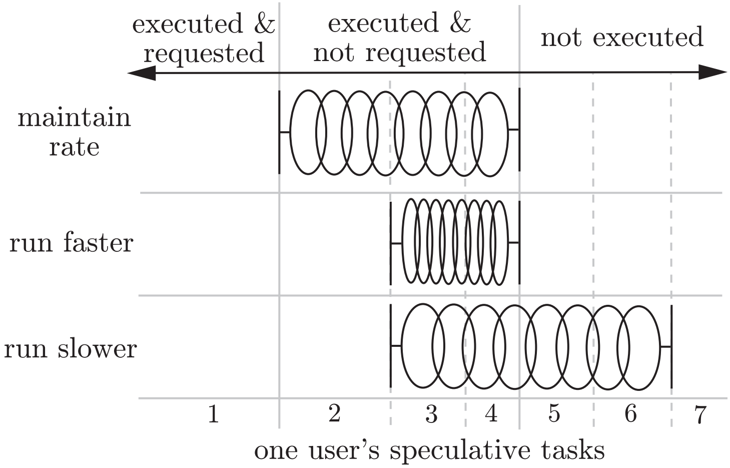


Figure 5.13: How feedback affects when the scheduler injects speculative tasks. After a user receives task output, the left side of a per-user spring moves to the next speculative task whose output has not been requested. After a task executes, the right side moves to the next unexecuted task. If the sides touch and the user requests task output, the user experiences visible response time. If the spring stretches too far, tasks stop being injected and resources go to other users. Across all users, the independent feedback loops dynamically determine to what extent such springs resist compression and stretching.

a scalable Internet service, Welsh et al. [2001] use a controller that, based on observed performance, adjusts the number of executing threads and lengths of event queues.

An alternative to control feedback could compute costs and benefits to determine whether a speculative task should enter the system. This analytic approach succeeded for prefetching disk requests [Patterson et al., 1995] (Chapter 2.5.3). However, building, validating, verifying, and maintaining this model is harder in the more dynamic settings under consideration: e.g., some disclosures are never needed; think time can be complex (for the web prefetching application, it is bimodal because often either one reads through a web page in its entirety or quickly decides that the page is not of interest); load is dependent on work issued outside of the independent schedulers' control (such as network conditions); and service time is highly variable. These uncertainties make it difficult to predict a speculative task's deadline and response time which this approach would need to make decisions.

5.10 Summary

This chapter introduced batchactive scheduling. People wish to batch their planning and submission of tasks and pipeline the consideration of needed task outputs with the execution of remaining tasks. Non-speculative schedulers present obstacles to this way of working, motivating my solutions. Using the knowledge that speculative tasks might be unneeded, batchactive schedulers intelligently order tasks toward maximizing human productivity, minimizing user resource costs, and promoting efficient use of server resources.

I began by defining a speculative task as one that the submitter does not know is required at the time of submission. While any task may be more or less speculative than another, the user only discriminates between speculative or not when submitting a task. For important application scenarios, users already know which of their tasks are speculative (Chapter 2.2).

Speculative tasks may be ordered as a list or unordered (meaning that execution order is irrelevant). General orders specified by a directed acyclic graph are possible but considered unlikely. The outputs of executed disclosed tasks are stored in an isolated location until they are requested by the user or canceled. No architectural changes to the non-speculative environment of the last chapter (Chapter 4.1) are needed.

I introduced the batchactive pricing mechanism which motivates users to disclose speculative work freely and deeply so that the scheduling policies can best meet user and resource provider performance and cost goals. Resources used by requested tasks — tasks whose outputs are known to be needed — are charged. Unrequested speculative tasks, which may have consumed resources, are not charged. In environments where resource usage is not directly charged to the user, performance may be driven down to the level of a non-speculative scheduler by abusive users who mark speculative work as demand work.

I defined visible response time which tracks the time that a user is waiting for a task's output. A task's visible response time may be less than its service time, because if the task had been disclosed it may have been chosen to run before being requested. The primary user scheduling goal is to minimize mean visible response time. In cases where resource usage is not directly charged, the goal is instead to minimize the variance of user requested resource usage. In comparing user costs between the batchactive and non-speculative pricing mechanisms, I track the ratio of resources used for needed tasks to all resources consumed by a user's speculative and non-speculative tasks. In comparing server revenue between pricing mechanisms, I measure requested load.

All batchactive policies use the knowledge that some tasks are speculative. Some may use estimates on whether and when a speculative task will be requested. I discussed techniques for predicting this information and determined that predicting task deadlines is impractical. An optimal algorithm for minimizing visible response time even with such information is unknown.

The implemented batchactive policies are two-tiered, consisting of a subpolicy for requested tasks and a subpolicy for disclosed tasks. The requested queue subpolicies are SRPT, FCFS, and user-requested-FB. The disclosed queue subpolicies include these and the following two novel ones: HRP, which favors users who are better speculators, and HRR, which favors users who have requested more work. I showed that emulating two-tiered batchactive scheduling by manipulating several existing types of schedulers is tedious.

I ended this chapter with a look at speculative scenarios beyond centrally scheduled processing resources which can be addressed using a feedback-based approach to adjust the rate at which speculative tasks are injected.

Quantus tremor est futurus,
Quando judex est venturus,
Cuncta stricte discussurus!
(author unknown), *Dies Irae, Missa pro defunctis*

6 Simulation results

Batchactive schedulers (Chapter 5) utilize their knowledge of which tasks are speculative to achieve better time- and cost-based scheduling metrics for both users and their resource provider (Table 5.1) than non-speculative schedulers (Chapter 4).

I measure speculative and non-speculative scheduling behavior with a discrete event simulator [Ball, 2004] that I wrote called `ba_sim` and that I describe in Chapter 7.1. Simulations tabulate metrics under a model of synthetic user behavior and task workloads and a single server.

I have not developed an analytic model to support simulation results. User behavior (Chapter 2.1) includes aspects (e.g., multiple tasks disclosed per user and tasks possibly being canceled) motivated by my target application scenarios (Chapter 2.2) which are difficult (Chapter 5.4.1) to faithfully analyze. Moreover, scheduling policies (Tables 4.3 and 5.3) have complexities (e.g., the learning aspect of HRP) also difficult to analyze. I discuss the methods I employed to increase the confidence in my simulator's correctness and I apply operational laws to a non-speculative simulation run to verify aspects of the simulator which have well-known analytic analogues in Chapter 6.1.4.

The results demonstrate improved metrics when users use batchactive schedulers compared to users who submit one needed task at a time or batches of speculative tasks to non-speculative schedulers. For example, FCFS × FCFS performs at least twice as well for about 20% of the wide-ranging simulated scenarios for mean visible response time relative to its non-speculative counterpart. (Recall that I notate two-tiered batchactive schedulers as *requested task subpolicy* × *disclosed task subpolicy*.) Moreover, visible response time can be improved without lowering visible task throughput and sometimes without lowering server revenue. Related, more users can be supported at the same mean visible response time.

I show that existing solutions to scheduling the important scenarios in Chapter 2.2 fall short and that batchactive solutions work well. First I describe the simulated model. I justify simulation parameters with reference to real tasks and studies of user behavior and discuss why I believe my simulator models reality sufficiently well for the scheduling comparisons I present and why I believe my simulator functions correctly. Following I present results comparing batchactive scheduling to non-speculative scheduling. The simplest compared policies are two-tiered FCFS and non-speculative FCFS. I proceed to novel disclosed queue policies, usage-based scheduling, and size-based scheduling. Before summarizing, I cover simulation details.

6.1 Simulation model

The simulator models users who cycle between submitting tasks to a single server, waiting for task output, and thinking about task output. On non-speculative schedulers, users may submit tasks belonging to their task sets one at a time or all at once, which I call interactive and batch usage of the system, respectively. (Intermediate levels of submission are not modeled. I justify these modeling extremes in Chapter 6.1.2.) On batchactive schedulers, users disclose all tasks belonging to their task sets at once, which I call batchactive usage of the system.

The simulated server runs tasks according to the scheduling policy under test. A task, which may be preempted according to the policy, completes when the simulated time that it ran meets its service time. Because most tasks under consideration (Chapter 2.2) are expected to last at least a minute, the simulated server assumes zero preemption overhead. (Preemption overhead in a real system may be caused by context switches or translation lookaside buffer flushes. These are orders of magnitude shorter than a minute.) Even if most tasks lasted only a few seconds, this overhead would only be an issue if batchactive scheduling caused more preemptions, which is not the case as shown in Table 6.6.

Parameterized simulations explore (Chapter 6.2) a variety of load, user behavior, and task characteristics. The following sections detail the simulation model and establish confidence in its operation.

6.1.1 Task submission and task output consumption cycle

Simulated users interact with the system in a way motivated by the work patterns described in Chapter 2.1. A number of users enter the system at the start of the simulation and plan speculative work as *task sets,*[1] which

[1] Real users may explore more than one hypothesis, submitting more than one task set at once. This ‘branching’ is not modeled, but I suspect it would provide a better advantage

are organized as lists[2] of a finite number of tasks (Chapter 5.2). List order reflects an any-time or iterative improvement task set (Chapter 2.1) or sequential tasks (Chapter 2.2.2).

With a non-speculative scheduler, simulated users *request* these tasks either as needed (i.e., one at a time) or all at once as described next in Chapter 6.1.2. With a speculative scheduler, users *disclose* these tasks and *request* them only when the users need their outputs. A simulated user receives task output after a requested task completes and considers the output for some *think time.*[3] Then the user may need the next task output, *cancel* remaining tasks and submit a new task set, or submit a new task set if the end of the current task set has been reached (i.e., users never depart). I do not simulate users who cancel only parts of their remaining task sets. Whole task set cancelation is performed because I believe it is simplest for the user, it restricts the number of simulation parameters, and it fits some application scenarios (Chapter 2.2). This task submission and task output consumption cycle, illustrated in Figure 6.1, determines the deadlines of speculative tasks whose outputs are eventually determined to be needed.

In queuing theory, an open system's tasks arrive independently (e.g., according to a Poisson process). A closed system has a constant number of users who each submit one task at a time and only submit the next task after the previous completes and an optional think time elapses. My simulation model can be considered a closed system with modification. Like a closed system, the simulator models a constant number of users. However, unlike a standard closed system, task arrivals can occur independently of server performance: speculating users each submit *multiple* speculative tasks, some of which may be *canceled* independent of the amount of service they receive, causing new task sets to be disclosed. I model users this way, instead of as a traditional closed system, because it better reflects how people work according to my findings (Chapter 2.1). It is likely that standard closed system operational laws can be enlarged to model these properties.

Generated user behavior ignores scheduling performance. In reality, lower mean visible response time might cause longer think time if more time spent considering task output instead of waiting for task output increases user

to batchactive scheduling: load would increase, hurting non-speculative schedulers that do not discriminate between needed and speculative work, and the user would be sated from output from any of his or her disclosed task sets.

[2] DAG ordered and unordered task sets (Chapter 5.2) are not simulated.

[3] Away periods are not simulated (Chapter 2.3.2). I believe they would provide a better advantage to batchactive scheduling than non-speculative scheduling because they would provide a greater opportunity for needed work to be prioritized over speculative work.

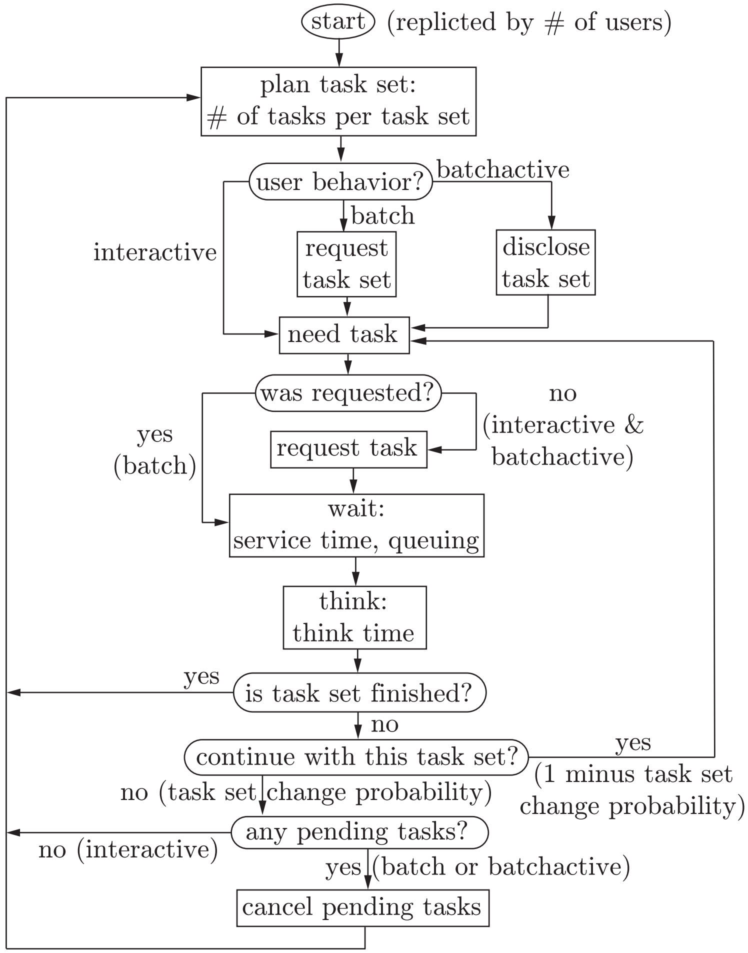


Figure 6.1: Flowchart of the model of user behavior. The circle is the start state, boxes are actions, and rounded boxes pose choices. Three types of users are modeled (Chapter 6.1.2): interactive, batch, and batchactive. Interactive and batch users use non-speculative schedulers while batchactive users use batchactive schedulers that allow task disclosure. In the following description, simulation parameters (Chapter 6.1.3) are italicized. This flowchart is replicated by a *number of users* which arrive at the start of a simulation and never depart. A user plans a task set with a *number of tasks per task set.* The user waits for a task based on the task's *service time* and queuing delays that result from the scheduler's policy. The user may wait less than the service time if the user was batch or batchactive and was able to submit the task before its output was needed. The user thinks for some *think time,* after which the user may decide to start a new task set based on his or her *task set change probability.* This flowchart is simplified for the different user types in Figures 6.2, 6.3, and 6.4 for interactive, batch, and batchactive users, respectively.

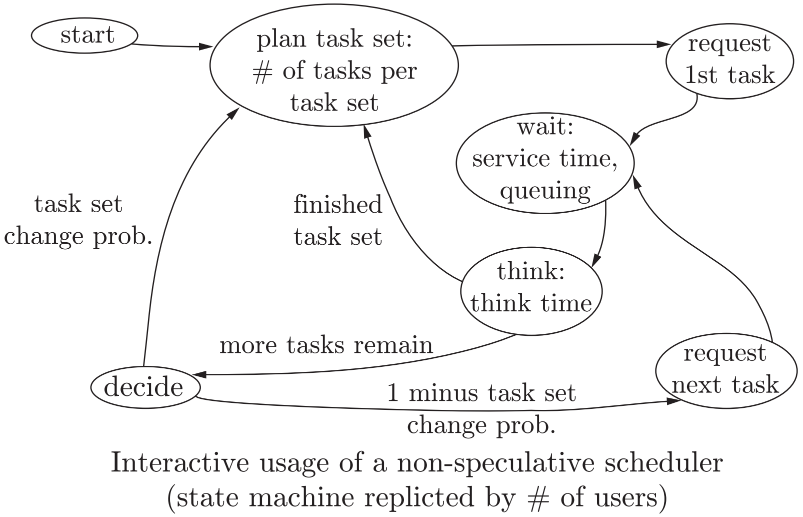

start
plan task set: # of tasks per task set
request 1st task
wait: service time, queuing
task set change prob.
finished task set
think: think time
more tasks remain
decide
request next task
1 minus task set change prob.
Interactive usage of a non-speculative scheduler (state machine replicted by # of users)

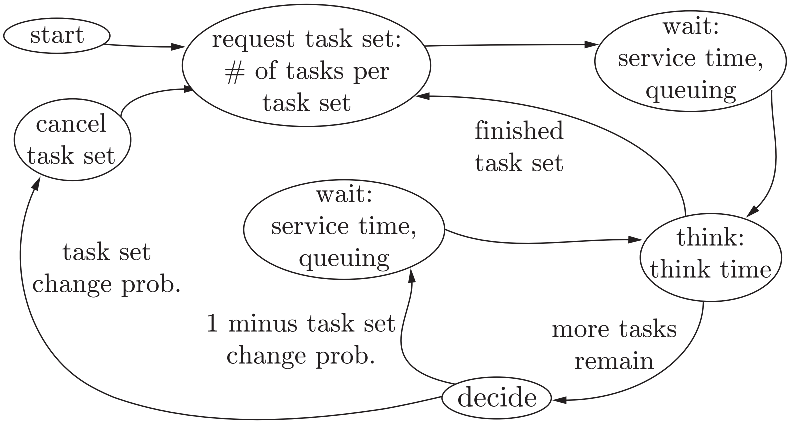
Figure 6.2: Interactive usage of a non-speculative scheduler. At the start of a simulation, a *number of users* arrive and never depart. Each follows the depicted state transitions. First, a user plans a task set made up of a *number of tasks per task set.* Then the user requests the first task, waits for its output based on the task's *service time* and queuing delays that result from the scheduler's policy. At this point, the user thinks about the task's output for some *think time.* If the task set is finished, the user plans the next task set. If not, the user decides, based on a *task set change probability,* whether to plan a new task set or request the next task in the task set. Simulation parameters (Chapter 6.1.3) were italicized.

fatigue, e.g. Yet my simulator would not change the user's think time in response to changing visible response time. A real world test (which is outside my scope) could answer whether this simplification significantly affects scheduling results.

6.1.2 Interactive v. batch v. batchactive usage

A cost-aware user would submit only one task at a time to a non-speculative scheduler to ensure being charged minimally. I call these users *interactive* in reference to the traditional definition in which one submits a task, waits for its output, then submits a new task based on the output as illustrated in Figure 6.2.

A user who is confident of needing all speculative tasks immediately, has abundant economic resources to pay for unneeded speculation, or is not directly charged for resource usage would instead submit entire sets of


start
request task set: # of tasks per task set
wait: service time, queuing
cancel task set
finished task set
wait: service time, queuing
think: think time
task set change prob.
1 minus task set change prob.
more tasks remain
decide

Batch usage of a non-speculative scheduler
(state machine replicated by # of users)

Figure 6.3: Batch usage of a non-speculative scheduler. At the start of a simulation, a *number of users* arrive and never depart. Each follows the depicted state transitions. First, a user requests all the tasks in a new task set made up of a *number of tasks per task set.* Then the user waits for the output of the first task based on the task's *service time* and queuing delays that result from the scheduler's policy. At this point, the user thinks about its output for some *think time.* If the task set is finished, the user requests a new task set. If not, the user decides, based on a *task set change probability,* whether to cancel the current task set and request a new task set or wait for the next task based on its *service time* and queuing delays. Here, the user may wait less than the service time if some of the user's think time occurred in parallel with the task's execution. Simulation parameters (Chapter 6.1.3) were italicized.

speculative tasks to a non-speculative scheduler. I call these users *batch* in reference to the traditional definition in which one submits multiple tasks at once as illustrated in Figure 6.3.

Between these extremes, a user using a non-speculative scheduler could submit a portion of his or her task set to meet a personal goal concerning the expected visible response times of tasks eventually determined to be needed v. the expected cost of consuming resources for tasks eventually determined not to be needed. However, it is difficult to define this goal and determine how many tasks to submit to meet it (Chapter 5.1). The utility of each task submission is determined by load, service time, think time, the probability of needing its output, and the value of time spent waiting for the output, and many of these considerations can be only predicted with uncertain accuracy.

I explore the simple case, the implications of behavioral extremes: my simulator models users using non-speculative schedulers who behave either interactively or in a batch manner; i.e., either submitting one task at a time or submitting entire task sets at a time.[4] Aspects of human psychology (Chapter 5.1.1) suggest that, when resource usage is directly charged, users would more likely be interactive. Each simulation of non-speculative scheduling involve users all of one type; i.e., all interactive or all batch.

When using a batchactive scheduler, users behave in a *batchactive* manner. Like batch users, batchactive users submit entire task sets at once; users would do so because the batchactive pricing mechanism does not penalize for unneeded speculation (Chapter 5.1.5). Like an interactive user, one task of a user's task set is *requested,* identifying to the scheduler the non-speculative task whose output the user is waiting for. Batchactive usage is illustrated in Figure 6.4.

Both batch and batchactive users will *cancel* task sets when received outputs indicate no need for additional outputs from those task sets. Since interactive users only submit tasks they need, they never cancel tasks.

6.1.3 Simulator parameters

The following simulation parameters realize the task submission and task output consumption cycle (Chapter 6.1.2): number of users, task set change probability, number of tasks per task set, service time, and think time. The choices of the task set change probability and the number of tasks per task set are made independently for each user to simulate users with different characteristics. After detailing the parameters, I justify the ranges of these parameters explored by the simulations (Chapter 6.2).

A constant *number of users* concurrently interact with a single server. Actual clusters have many nodes, some faster than others. The relative benefits of batchactive scheduling from my single server simulations should persist with multiple nodes. The number of users is varied across simulations. All users enter the system at the start of the simulation and never depart.

The *task set change probability* is the probability that, after considering a task's output, a user will cancel his or her current task set and submit a new task set. (If there are no more outstanding tasks in the task set, the user automatically submits a new task set after some think time.) Each user is assigned a task set change probability from a distribution. By assigning randomly chosen probabilities to each user, instead of having all users share the same probability, the simulator models users who are more or less certain

[4] I considered how partial task set submission to a non-speculative scheduler could be automated with a frontend that throttles speculative task submissions in Chapter 5.9.2.

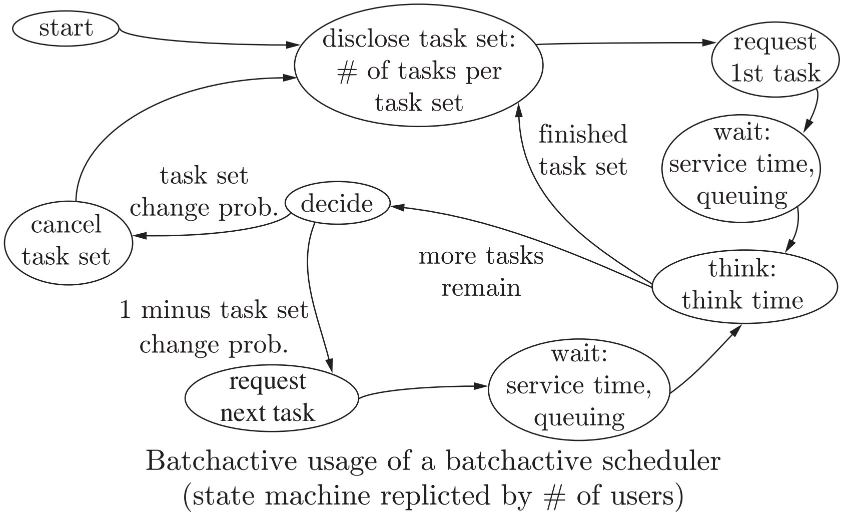


Figure 6.4: Batchactive usage of a batchactive scheduler. At the start of a simulation, a *number of users* arrive and never depart. Each follows the depicted state transitions. First, a user discloses all the tasks in a new task set made up of a *number of tasks per task set.* Then the user requests the first task, waits for the output of the task based on its *service time* and queuing delays that result from the scheduler's policy. At this point, the user thinks about its output for some *think time.* If the task set is finished, the user discloses a new task set. If not, the user decides, based on a *task set change probability,* whether to cancel the current task set and request a new task set or wait for the next task based on its *service time* and queuing delays. Here, the user may wait less that the service time if some of the user's think time occurred in parallel with the task's execution. Simulation parameters (Chapter 6.1.3) were italicized.

about whether they will need their speculative work. Values are chosen from a continuous uniform distribution[5] whose lower bound is always 0 and whose upper bound is varied across simulations. A small upper bound simulates users who disclose speculative tasks they almost certainly need. A large upper bound simulates a wider range of users, including those who frequently cancel task sets. For example, if the upper bound is set to 0.2, then users will be created with task set change probabilities ranging between 0 and 0.2. If a specific user is created with a task set change probability of 0.1, then after thinking about the output of each needed task, 10% of the time the user will cancel and issue a new task set.

The *number of tasks per task set* dictates how many speculative tasks make up task sets submitted by users at the start and throughout a simulation. Like the task set change probability, each user is assigned a random number of tasks per task set from a distribution so that the simulator is able to model users with unique characteristics; here, users who speculate to varying depths. Values are chosen from a continuous uniform distribution whose lower bound is always 1, reflecting no disclosure, and whose upper bound is varied across simulations. A small upper bound simulates users who disclose shallowly; scientists planning up to five or so experiments ahead. A large upper bound simulates a wider range of users, including those disclosing hundreds of tasks deep, reflecting an automated process working on behalf of a user searching high-dimensional spaces. For example, if the upper bound is set to 20, then users will be created with numbers of tasks per task set ranging between 1 and 20. If a specific user is created with a number of tasks per task set of 10, then all his or her task sets will consist of 10 speculative tasks.

One way to reason about speculation is to consider the probability that a task set is canceled before all its tasks are needed. This *degree of speculation* can be calculated as a function of a task set change probability, p, and a number of tasks per task set, n. Because each user needs the first task in

[5] A continuous uniform distribution [Weisstein, 2004f] has constant probability over some range. The probability density function and cumulative distribution function of a continuous uniform distribution on the interval $[a, b]$ are

$$P(x) \overset{\text{def}}{=} \begin{cases} 0 & \text{for } x < a, \\ \frac{1}{b-a} & \text{for } a \le x \le b, \\ 0 & \text{for } x > b, \text{ and} \end{cases} \qquad D(x) \overset{\text{def}}{=} \begin{cases} 0 & \text{for } x < a, \\ \frac{x-a}{b-a} & \text{for } a \le x \le b, \\ 1 & \text{for } x > b. \end{cases}$$

The expected value of this distribution is $\frac{1}{2}(a+b)$ and its variance is $\frac{1}{12}(b-a)^2$. This distribution exhibits an increasing failure rate; the longer a phenomenon has existed, the more likely it will terminate.

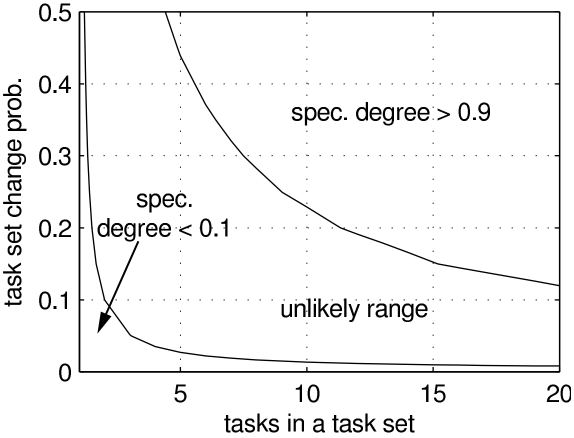


Figure 6.5: A contour plot showing how the number of tasks per task set and the task set change probability affect whether a user will cancel his or her current task set. Speculating and non-speculating users will tend toward the upper-right and lower-left of the space, respectively.

his or her task set, the probability that the user will cancel the task set is $1-(1-p)^{n-1}$. Assuming that users either practice speculation, with a degree of speculation exceeding perhaps 0.9, or do not practice speculation, with a degree less than 0.1, then the combinations of number of tasks per task set and task set change probabilities for each user will not be in the middle region of the space depicted in Figure 6.5.

Service time dictates the sizes of tasks. Each task is assigned a random service time from a distribution regardless of which user submitted the task. Values for most experiments are chosen from an exponential distribution[6] whose mean (the inverse of the distribution's rate parameter λ) is varied

[6] An exponential distribution [Weisstein, 2004a] has a tail that drops by a constant factor at constant intervals. The probability density function and cumulative distribution function of an exponential distribution given rate $\lambda > 0$ and $x \geq 0$ are

$$\begin{aligned} P(x) &\overset{\text{def}}{=} \lambda e^{-\lambda x} \text{ and} \\ D(x) &\overset{\text{def}}{=} 1 - e^{-\lambda x}. \end{aligned}$$

The expected value of this distribution is $\frac{1}{\lambda}$ and its variance is $\frac{1}{\lambda^2}$.

The exponential distribution is memoryless, also known as a constant failure rate (Chapter 4.5.1), meaning that $P(X > s+t \mid X > s) = P(X > t)$ for any s and $t \geq 0$. That is to say, the probability that a task runs for at least t more seconds before terminating given that the task has already run for s seconds is the same as the probability that the task runs at least t seconds independent of s — history does not matter.

across simulations. For example, if the mean is set to 1000, then the mean service time for all tasks in a simulation will tend toward 1000.

The pervasiveness of the exponential distribution in the literature (including for modeling task sizes in a gang scheduling simulator from Feitelson and Jette [1997]) — due to the analytic tractability of its memoryless property and the (diminishing) belief that important phenomena are distributed according to it (whereas increasing evidence shows that the Pareto distribution, described next, fits better) — motivates its use in most of my simulations, enabling the broadest understanding of the results I present, especially compared to distributions with parameters whose settings may be hard to justify.

For some runs, service time is chosen from a Pareto distribution.[7] Pareto distributions with low α are found to model closely many phenomena within and beyond computing [Harchol-Balter, 1999; Crovella, 2000]. Harchol-Balter and Downey [1997] have shown that a Pareto distribution with $0.8 < \alpha < 1.2$ models Unix task sizes well, leading to their rule of thumb of setting $\alpha = 1$ for modeling task service time.

Pareto distributions whose α parameters are less than or equal to 2 are observed often. Such a Pareto distribution is called heavy-tailed, meaning that its tail follows a power law with low exponent. These heavy-tailed distributions more strongly exhibit decreasing failure rates (Chapter 4.5.1) than non-heavy-tailed Pareto distributions. Moreover, the majority of mass is concentrated in a small subset of observations. For service time, these properties imply that the longer a task has run, the longer it can be ex-

[7] A Pareto distribution [Weisstein, 2004d] has a tail that drops according to a power law. The probability density function and cumulative distribution function of a Pareto distribution (also known as a power-law, hyperbolic, or double-exponential distribution [Paxson and Floyd, 1994]) are defined for $x \geq b$ and $\alpha > 0$ as $\frac{\alpha b^\alpha}{x^{\alpha+1}}$ and $1 - (\frac{b}{x})^\alpha$, respectively. For convenience, b is set to 1. Thus, for $x \geq 1$,

$$P(x) \overset{\text{def}}{=} \frac{\alpha}{x^{\alpha+1}} \text{ and}$$
$$D(x) \overset{\text{def}}{=} 1 - \left(\frac{1}{x}\right)^\alpha.$$

To ensure that sufficiently small events can be represented, given the restriction on x, a small unit such as seconds can be employed.

The expected value of a Pareto distribution for $0 \leq \alpha \leq 1$ is infinite. When $\alpha > 1$, the expected value is $\frac{\alpha}{\alpha-1}$. The variance of a Pareto distribution for $0 \leq \alpha \leq 2$ is infinite. When $\alpha > 2$, the variance is $\frac{\alpha}{(\alpha-1)^2(\alpha-2)}$. This distribution exhibits a decreasing failure rate; the longer a phenomenon has existed, the less likely it will terminate. Put another way, if a phenomena is Pareto distributed with an α parameter of 1, then there is a 50% probability that an instance existing for x seconds will exist for another x seconds.

pected to run, and that most tasks are small, but most service time is taken by a small number of large tasks. [Crovella, 2000] As α tends toward 0, these heavy-tailed properties are more pronounced and vice-versa as α tends toward 2 [Harchol-Balter, 2003b].

These properties of heavy-tailed Pareto distributions complicate simulation and analysis. When sampling random variables from a heavy-tailed Pareto distribution, very large observations occasionally occur, a consequence of the infinite moments [Weisstein, 2004c] of this distribution. A large number of samples from the distribution must be made — so that very large observations on the tail are represented — before a simulation reaches steady state. Further, the infinite moments prevent the application of useful formulas such as the Pollaczek-Khinchin formula which shows that the expected time-in-queue for a task taken from a general service time distribution with Poisson interarrival times on a single server employing the FCFS policy is $\frac{\rho}{1-\rho} \cdot \frac{E[S^2]}{2E[S]}$, where ρ is load and $E[S]$ is the expected service time. [Crovella and Lipsky, 1997]

A truncated version of the Pareto distribution which reflects that measured data has minimum and maximum observations (e.g., task sizes) is the Bounded Pareto Distribution.[8] This distribution has finite moments, simplifying simulation and analysis.

Think time dictates the time that users consider task outputs. Each time a task's output is delivered to a user, a think time value is chosen randomly from a distribution regardless of which user received the output. Values for most experiments are chosen from an exponential distribution whose mean is varied across simulations. Values for some experiments are chosen from a Pareto distribution.

The parameter ranges listed in Table 6.1, unless otherwise specified, were

[8] A bounded Pareto distribution [Crovella et al., 1997; Harchol-Balter, 2003b] is a Pareto distribution with minimum and maximum observations. Its tail drops according to a power law in the range of observations. The probability density function and cumulative distribution function of a Bounded Pareto distribution are defined for $k \leq x \leq p$ where k is the minimum observation, p is the maximum observation, and $\alpha > 0$ as

$$P(x) \stackrel{\text{def}}{=} \frac{\alpha k^{\alpha}}{1-(k/p)^{\alpha}} x^{-\alpha-1} \text{ and}$$
$$D(x) \stackrel{\text{def}}{=} \frac{1-(k/x)^{\alpha}}{1-(k/p)^{\alpha}}.$$

The jth moment of this distribution (which enables one to compute its expected value and variance) for $\alpha \neq j$ is $\frac{\alpha k^{\alpha}(k^{j-\alpha}-p^{j-\alpha})}{(\alpha-j)(1-(k/p)^{\alpha})}$. Although the upper bound gives this distribution an increasing failure rate, when $k \ll p$ many properties of the Pareto distribution, such as high variability, are maintained.

parameter	range	inc.	samples
number of users	1 to 16	3	6
task set change prob.	0.0 to 0.0–0.4 (uni.)	0.1	5
# of tasks per task set	1 to 1–21 (uni.)	5	5
service time (s)	20 to $3,620$ (exp.)	720	6
think time (s)	20 to $18,020$ (exp.)	3600	6

Table 6.1: The parameter ranges used in simulating users and tasks for the batchactive improvement results reported in Chapter 6.2. These ranges were motivated by the speculative scenarios in Chapter 2.2. A uniform distribution (uni.) is described by 'lower bound (a) to upper bound (b),' where the upper bound is specified by a range varied across runs. An exponential distribution (exp.) is described by its mean ($1/\lambda$), where the mean is specified by a range varied across runs. An increment (inc.) determines the distance between each sample in the range of its parameter.

used for the batchactive improvement results reported in Chapter 6.2. ('Improvement' is formalized in Chapter 6.2.3.) For each parameter, I sampled several points in its range. All told, each user behavior and scheduler combination was evaluated against $5,400$ selections of parameters, the product of the number of samples for each parameter.

The choice of simulation parameter values is key to arguing for batchactive scheduling. I can make batchactive scheduling look arbitrarily better than common practice by selecting parameter values that highlight its strengths. However, this would not be a convincing argument. Instead, I have chosen parameter ranges that not only include what I believe to be reasonable uses of speculation for the target applications (Chapter 2.2), but also ranges that include little or no speculation:

- The range of the number of users was chosen, based on the other parameter choices, to provide minimal resource contention among users at the lower bound and to consume all of the simulated single server's resources at the upper bound.

- The upper bound of the task set change probability ranges from modeling a user who always needs his or her speculative tasks (0%) to one who cancels his or her task sets 40% of the time after considering a single task's output. Bubenik and Zwaenepoel [1989] found that 39% of speculative application rebuilds were canceled (Chapter 2.3.1).[9] Along with the following parameter, the range of task set change probabil-

[9] Because these rebuilds were started by an automatic process, I consider this figure to be an upper bound for the cancelation of user-initiated task sets.

ities cover the regions of small and large degrees of speculation from Figure 6.5.

- The upper bound of the number of tasks per task set ranges from no disclosure (1), modeling a user who cannot plan ahead, to a little over twenty disclosures, modeling a user who uses domain-specific knowledge to make small to medium-sized computational plans. If the number of tasks per task set is too large, the user will be constantly interrupted with new task outputs. If too small, there is no opportunity to pipeline task execution with user think time. These considerations motivate the range of task set sizes I explore.

- Mean service time, which varies from one third of a minute to about one hour, is based on BLAST DNA similarity searches [Giddings and Knudson, 2004; Biowulf, 2004], film frame rendering [Hillner, 2003; Epps, 2004; Lokovic, 2004], and the wide-ranging exploratory searches and parameter studies detailed in Table 4.4.

- Mean think time, which varies from one third of a minute to roughly five hours, reflects a user who can make a quick decision about a task's output to one who needs to graph, ponder, or discuss output with colleagues. Evidence of think time in actual workloads was cited in Chapter 2.3.1.

Besides the improvement results, I also present results from simulations exploring the range of one parameter at a time. These sensitivity analyses, which look at slices of the simulator's parameter space, determine to what extent each parameter affects results and help identify best- and worst-case environments for batchactive scheduling. For these results, all parameters except for the single varying parameter were fixed at the values in Table 6.2 unless otherwise noted.

6.1.4 Determining model and simulator correctness

The conclusions of this thesis come from my interpretation of results generated by my `ba_sim` simulator. My interpretation (Chapter 6.2.2) can be evaluated by the reader. The correctness of my (or any) simulation results can be argued through model validation and verification. Model validation is the 'substantiation that a computerized model within its domain of applicability possesses a satisfactory range of accuracy consistent with the intended application of the model,' and model verification is 'ensuring that

parameter	setting
number of users	8
task set change probability	0.0 to 0.2 (uniform)
number of tasks per task set	1 to 15 (uniform)
service time (s)	600 (exponential)
think time (s)	$6,000$ (exponential)

Table 6.2: The fixed parameters used in the sensitivity analyses in Chapter 6.2. These values fall within the ranges in Table 6.1 used for the summarizing improvement results. For each sensitivity analysis, all but one parameter were held constant at these values. The range, increment, and number of samples for the varied parameter is evident in the horizontal axes of the parameter study figures. A uniform distribution is described by 'lower bound (a) to upper bound (b).' An exponential distribution is described by its mean ($1/\lambda$).

the computer program of the computerized model and its implementation are correct.' [Schlesinger and others, 1979]

The validation of the simulation model follows some recommendations of Sargent [1999]. Most importantly, I asked people knowledgeable about scheduling whether my model was reasonable, a technique known as *face validity.* Related are *historical methods*, which is using assumptions that most hold, such as exponential and Pareto task size distributions. I also used *operational graphics* to visualize that various performance measures over time appeared reasonable (such as Figures 6.66 and 5.9). I created *degenerate tests* to confirm that certain parameters led to expected pathological behavior. *Extreme condition tests* showed that the output was plausible for unlikely combinations of parameters (Figures 6.20 and 6.31). I made several runs of the simulation with the same parameters but different random number seeds for generating random variables to confirm that the variability was sufficiently small to make the results dependable, a technique known as *internal validity,* as shown in Chapter 6.3.2.

Model verification is concerned with determining that the simulation was implemented correctly [Sargent, 1999]. Model verification for simulators written in general-purpose computer programming languages (as opposed to simulation languages), such as mine, involves the use of the software engineering techniques discussed in Chapter 7.1.3.

Further, I verify some simulation data using analytic techniques. Little's Law and the Utilization Law (well-known 'operational laws' which make no assumption on service orders or service time distributions[10]) were derived for standard open and closed systems and thus do not apply to my simulation model which is an extended closed system that includes multiple submitted tasks per user which have the chance of being canceled (Chapter 6.1.1). However, they apply to a special case in which task sets consist of only one task and the task set change probability for each user is always 0, i.e., tasks are not speculative. When running `ba_sim` in this manner, the closed system operational laws apply directly and verify the integrity of the simulation. Little's Law for a closed system states that $E[R] = N/X - E[Z]$, where $E[R]$ is the expected (mean) response time, N is the number of users, and $E[Z]$ is the expected (mean) user think time. The Utilization Law states that $U = XE[S]$, where U is load and $E[S]$ is the expected (mean) task service time. [Harchol-Balter, 2003b]

I ran `ba_sim` for a two-week simulation (with two warmup days removed) of eight users issuing non-speculative task sets with only one task per task set, with task service time exponentially distributed with mean 300 s and user think time exponentially distributed with mean $1,600$ s. The differences between the theoretical and simulated metrics[11] were less than one percent and are presented in Table 6.3. This test shows accurate results when exercising the simulator with parameter selections simulating task set sizes of 1 and tasks with a 0% probability of task set cancelation, providing confidence in the simulator's operation when parameter selections expand to larger task sets made up of tasks which might be canceled.

Finally, I show a sample `ba_sim` run with no warmup period of pertinent user, task, resource, and scheduler events followed by metrics calculated by the simulator. This is a 12-hour simulation of two users behaving in a batchactive manner on the FCFS × FCFS two-tiered batchactive scheduler. The reader may verify that the metrics are correct, providing confidence that the metrics output in Chapter 6.2 are also correct. Task identifiers are with respect to the submitting user. Identifiers are numbered from 0. The

[10] Operational laws apply for any ergodic system. An ergodic system is positive recurrent (i.e., the system probabilistically restarts itself, such as by having the possibility of being emptied of tasks), aperiodic (i.e., the system state, such as the number of tasks in the system, does not depend on time step), and irreducible (i.e., it is possible to get from any state, where the state could be the number of tasks in the system, to any other state, meaning that the initial state is irrelevant).

[11] These metrics may be reproduced by running `ba_sim` with the command-line arguments '`-v 1382400 -n 8 -b uniform:0,0 -c uniform:1,1 -d exponential:300 -e exponential:1600`'. For use by the operational laws, the reported throughput must be scaled to tasks per second by dividing the throughput by the number of seconds in two weeks $(1,209,600)$.

	theoretical	simulation
mean resp. time	1026.710	1012.618
load	0.918618	0.917004

Table 6.3: Non-speculative verification using operational laws. I compare the metrics output by `ba_sim` with theoretical metrics (from Little's Law and the Utilization Law) for a simulation of task sets made up of one non-speculative task each. (Thus the non-speculative metrics and batchactive metrics are equivalent; specifically, mean visible response time and mean response time are equal and requested load and load are equal.) I found less than one percent difference for mean response time and less than one tenth of one percent difference for load.

abbreviation 'st' means 'service time,' while 'req q' means 'requested queue,' and 'dis q' means 'disclosed queue.'[12]

```
user 0 created at time 0.000
user 1 created at time 0.000
user 0 planned a task set of 5 task(s) at time 0.000
user 0 disclosed task 0 (st: 5279.514) at time 0.000
user 0 disclosed task 1 (st: 8314.654) at time 0.000
user 0 disclosed task 2 (st: 2982.928) at time 0.000
user 0 disclosed task 3 (st: 4426.236) at time 0.000
user 0 disclosed task 4 (st: 2656.431) at time 0.000
user 1 planned a task set of 2 task(s) at time 0.000
user 1 disclosed task 0 (st: 4506.721) at time 0.000
user 1 disclosed task 1 (st: 17292.122) at time 0.000
user 0 needed (requested) task 0 (remain: 5279.514) at time 0.000
user 1 needed (requested) task 0 (remain: 4506.721) at time 0.000
sched chose user 0's task 0 (remain: 5279.514) from req q at time 0.000
resource running user 0's task 0 for at most 5279.514 at time 0.000
resource going idle after running user 0's task 0 at time 5279.514
user 0's task 0 executed at time 5279.514
user 0 started thinking about task 0 for 8602.712 at time 5279.514
sched chose user 1's task 0 (remain: 4506.721) from req q at time 5279.514
resource running user 1's task 0 for at most 4506.721 at time 5279.514
resource going idle after running user 1's task 0 at time 9786.234
user 1's task 0 executed at time 9786.234
user 1 started thinking about task 0 for 33219.854 at time 9786.234
sched chose user 0's task 1 (remain: 8314.654) from dis q at time 9786.234
resource running user 0's task 1 for at most 8314.654 at time 9786.234
user 0 needed (requested) task 1 (remain: 8314.654) at time 13882.226
resource going idle after running user 0's task 1 at time 13882.226
sched chose user 0's task 1 (remain: 4218.663) from req q at time 13882.226
resource running user 0's task 1 for at most 4218.663 at time 13882.226
resource going idle after running user 0's task 1 at time 18100.889
user 0's task 1 executed at time 18100.889
```

[12] This output may be reproduced by compiling `ba_sim` with the `BA_SIM_INSPECT` macro in `ba_sim.h` set to `1` and running `ba_sim` with the command-line arguments '`-x -r 2742755273 -v 43200 -n 2 -c uniform:2,5 -d exponential:7200 -e exponential:10800`'.

```
user 0 started thinking about task 1 for 108.268 at time 18100.889
sched chose user 0’s task 2 (remain: 2982.928) from dis q at time 18100.889
resource running user 0’s task 2 for at most 2982.928 at time 18100.889
user 0 needed (requested) task 2 (remain: 2982.928) at time 18209.157
resource going idle after running user 0’s task 2 at time 18209.157
sched chose user 0’s task 2 (remain: 2874.659) from req q at time 18209.157
resource running user 0’s task 2 for at most 2874.659 at time 18209.157
resource going idle after running user 0’s task 2 at time 21083.816
user 0’s task 2 executed at time 21083.816
user 0 started thinking about task 2 for 5911.972 at time 21083.816
sched chose user 0’s task 3 (remain: 4426.236) from dis q at time 21083.816
resource running user 0’s task 3 for at most 4426.236 at time 21083.816
resource going idle after running user 0’s task 3 at time 25510.052
user 0’s task 3 executed at time 25510.052
sched chose user 0’s task 4 (remain: 2656.431) from dis q at time 25510.052
resource running user 0’s task 4 for at most 2656.431 at time 25510.052
user 0 needed (requested) task 3 (remain: 0.000) at time 26995.789
user 0 started thinking about task 3 for 4378.416 at time 26995.789
resource going idle after running user 0’s task 4 at time 28166.483
user 0’s task 4 executed at time 28166.483
sched chose user 1’s task 1 (remain: 17292.122) from dis q at time 28166.483
resource running user 1’s task 1 for at most 17292.122 at time 28166.483
user 0 needed (requested) task 4 (remain: -0.000) at time 31374.205
user 0 started thinking about task 4 for 8632.173 at time 31374.205
user 0 planned a task set of 5 task(s) at time 40006.378
user 0 disclosed task 5 (st: 3410.037) at time 40006.378
user 0 disclosed task 6 (st: 4025.403) at time 40006.378
user 0 disclosed task 7 (st: 4874.664) at time 40006.378
user 0 disclosed task 8 (st: 6677.045) at time 40006.378
user 0 disclosed task 9 (st: 14983.982) at time 40006.378
user 0 needed (requested) task 5 (remain: 3410.037) at time 40006.378
resource going idle after running user 1’s task 1 at time 40006.378
sched chose user 0’s task 5 (remain: 3410.037) from req q at time 40006.378
resource running user 0’s task 5 for at most 3410.037 at time 40006.378
user 1 planned a task set of 2 task(s) at time 43006.088
user 1 canceled task 1 (remaining: 5452.227) at time 43006.088
user 1 disclosed task 2 (st: 21129.885) at time 43006.088
user 1 disclosed task 3 (st: 1182.141) at time 43006.088
user 1 needed (requested) task 2 (remain: 21129.885) at time 43006.088
resource going idle after running user 0’s task 5 at time 43006.088
sched chose user 0’s task 5 (remain: 410.328) from req q at time 43006.088
resource running user 0’s task 5 for at most 410.328 at time 43006.088
resource going idle after running user 0’s task 5 at time 43200.000

mean visible response time: 3693.178
mean visible slowdown: 0.774
visible task throughput: 6
number of deadlines met: 2
load: 1.000
requested load: 0.726
```

The design of my simulated model was chosen as the simplest model (with the fewest parameters needed to be justified) to reflect the speculative behavior in actual application scenarios (Chapter 2.2). The only unimpeachable test to determine batchactive benefits would be a deployment on real systems with actual users, serving as validation of the ideas and simulation model and verification of the simulator implementation. After enough use, one could survey users to obtain a comparative reaction against a non-speculative cluster scheduler. Doing so is outside my scope.

6.2 Scheduling policy comparison

I compare the performance of the non-speculative schedulers FCFS, user-FB, and SRPT (Table 4.3) against two-tiered batchactive schedulers. The two-tiered batchactive schedulers' requested queues are serviced by FCFS, user-requested-FB, and SRPT and their disclosed queues are serviced by FCFS, SRPT, HRP, HRR, and RFCFS (Table 5.3). Not every combination of requested and disclosed queue subpolicies address clear scheduling goals, and thus not every combination is evaluated. The choices of scheduling subpolicies in the following simulations were made to answer the following questions:

- How does the simplest, most easily deployable two-tiered batchactive scheduler, FCFS × FCFS, compare to the simplest size-agnostic non-speculative scheduler FCFS? (Chapter 6.2.4.)
- Can the novel disclosed queue subpolicies, HRP and HRR, which discover and employ historical user patterns, better schedule speculative tasks? (Chapter 6.2.5.)
- How does a batchactive scheduler with the best disclosed queue subpolicy discovered in the previous experiment compare against non-speculative FCFS? (Chapter 6.2.6.)
- When usage-based policies are employed, to what extent does batchactive scheduling outperform non-speculative scheduling? (Chapter 6.2.7.)
- How does size-aware scheduling based on SRPT affect the comparison of batchactive and non-speculative scheduling? (Chapter 6.2.8.)
- What is the potential performance improvement for using the impractical RFCFS disclosed queue subpolicy in a batchactive scheduler compared to the best performing practical batchactive scheduler? (Chapter 6.2.9.)

First I review the reported metrics. Then I list the central conclusions from the results. Following I explain the graph formats and present the simulation data.

6.2.1 Reported metrics

The tabulated metrics include mean visible response time, mean visible slowdown, visible task throughput, variance of visible response time, the number of deadlines met, variance of user requested resource usage, mean scaled billed resources, (total) load, requested load, and uncharged load (Table 5.1).[13] For each run, these metrics were tabulated over two weeks of simulated time (making this a terminating or finite-horizon simulation) after two warmup days were ignored (Chapter 6.3.1). The task service time and user think time parameters are given in seconds (Tables 6.1 and 6.2). For consistency, metrics based on the time that a user was waiting for task output and metrics based on the resource time consumed by a task are also given in seconds.

Mean visible response time and mean visible slowdown are the main time-based metrics. Visible task throughput and variance of visible response time are reported to confirm that they are not worsened when other metrics improve with batchactive scheduling. The number of deadlines met is reported to see how often users receive task outputs immediately upon needing them. When resources are not directly charged, scheduling policies sometimes seek to limit resource abuse by lowering the variance of user requested resource usage in lieu of reducing time-based metrics.

Mean scaled billed resources and requested load are the cost-based metrics. The first shows how much the average user is charged for consuming unneeded resources. Before taking the average, the amount of billed resources for each user is scaled by the user's needed tasks' resource requirements. The quantity is always minimum (i.e., 1) under batchactive scheduling because users will disclose (not request) speculative tasks. The quantity is also minimum when users do not speculate, and thus comparisons against a set of only interactive users are not made. However, a speculating user using

[13] Although these metrics were defined in the chapter on batchactive scheduling (Chapter 5.2), they are also applicable to non-speculative scheduling by analogy to non-speculative metrics (Table 4.1). Mean visible response time and mean visible slowdown are the average time between a user needing a task and its completion and the average of each visible response time scaled by its corresponding task size, respectively. Visible task throughput counts the only the needed tasks that have completed. (A requested task under a speculative scheduler is always counted and a requested task under a non-speculative scheduler is counted only if the requesting user eventually needed its output.) Variance of visible response time reflects the differences in visible response times. The number of deadlines met is always 0 for interactive users because these users never submit tasks before needing their outputs (viz., these users never request speculative tasks). In non-speculative scheduling, because a task only runs when requested, the variance of user resource usage is the variance of user requested resource usage and load is requested load.

a non-speculative scheduler — the example user and batch user described in Chapters 2.1 and 6.1.2, respectively — is charged for all resource usage. Therefore, I only present mean scaled billed resources for simulations involving batch users. A resource provider's revenue is proportional to billed (requested) load (Chapters 4.2 and 5.1). In a batchactive scheduler, some load may be made up of disclosed tasks, leading to two additional metrics reported to better understand batchactive behavior: (total) load and uncharged load, the load taken by speculative tasks eventually known to not be needed.

6.2.2 Central conclusions

Here I list the conclusions I have drawn from the simulation results shown next.

- Batch usage delivers better time-based metrics than interactive usage in some cases and interactive usage does better than batch usage in other cases. This holds when varying nearly every simulation parameter. But batchactive does at least as well for time-based metrics, often better, than both uses of non-speculative scheduling in every case, including the 'cross-over' cases where interactive and batch usage perform the same. (Considerations of cost are discussed below.)

  Thus, *with a batchactive system, users do not need to decide how aggressively to submit speculative work:* they may disclose all work not known to be needed to obtain these time-based improvements. (Figures 6.10, 6.17, 6.22, 6.45, 6.47, 6.50, 6.55, 6.56, and 6.58.)

  Batch usage of a non-speculative scheduler is suited to few users because execution time and think time are pipelined and load is sufficiently low that one's speculative but unneeded tasks do not overly interfere with needed tasks. Interactive usage of a non-speculative scheduler is suited to many users because the server is always busy with requested tasks.

  Batchactive scheduling is better than batch usage of a non-speculative scheduler under many users because requested tasks never wait for speculative tasks; it is better than interactive non-speculative scheduling under few users because it fills idle time with speculative tasks.

  The degree of speculation (a function of the task set change probability and the number of tasks per task set) does not affect the time- and cost-based metrics of interactive users as these users do not submit speculative tasks. There is a greater dependence on the degree of speculation with batch users compared to batchactive users for time-based metrics because batchactive schedulers avoid speculative tasks when requested work exists. (Figures 6.17, 6.19, 6.22, 6.24, 6.47, and 6.58.)

For any given run, batchactive scheduling simultaneously provides better mean visible response time and visible throughput compared to non-speculative scheduling. Batchactive scheduling also simultaneously provides better mean visible response time and requested load (as described below, there are more cases in which batch usage of non-speculative scheduling provides better requested load, but those cases offer dismal mean visible response time). *Latency-sensitive users will not push traditional schedulers into regions of high billed load because, at those levels of revenue, visible response times are too high.* The latency threshold for batchactive scheduling, in contrast, is better. (Figures 6.15, 6.16, 6.46, and 6.57.)

Batchactive usage of a speculative scheduler adapts across a range of task and user characteristics, often beating any usage of a non-speculative scheduler. The more constrained resources are (i.e., if there are many users or many tasks per user), the more important it is to give priority to requested work. The less work, the more important it is to pipeline work with a user's think time. Batchactive scheduling accomplishes both.

– *Batchactive scheduling provides better performance compared to both* FCFS- *and usage-based batch scheduling.* Some comparisons of batchactive scheduling are against batch usage of FCFS and batch usage of user-FB. Recall that user-FB (Chapter 4.5.3), a variant of decay usage, looks at the total amount of resources consumed by a user. When a scheduling decision is to be made, user-FB selects the task from the user that has used the fewest resources at the time the decision is made.[14] See Chapter 4.6.3 for the reasons why either FCFS or user-FB are used in practice.

  Among non-speculative scheduling, results show that batch users on user-FB obtain better mean visible response time than batch users on FCFS, which is expected from the characteristics of these schedulers

[14] This common policy is an approximation to a suggested policy which selects tasks in a round-robin fashion across task sets (RRTS) whose motivation is to execute tasks in the order in which users will need task outputs. Because user-FB approximates the RRTS scheduling order, RRTS is not discussed further.

described in Chapter 4.6.3. (Figures 6.10 and 6.50, which have different vertical axis scales.)

Still, batchactive scheduling, such as user-requested-FB × HRP or FCFS × HRP, performs better than batch usage of user-FB (Figures 6.50 and 6.45, which have different vertical axis scales). The batchactive configuration performs better because it executes known-needed tasks first. The user-FB policy will execute disclosed tasks that will be canceled even when known-needed tasks exist. For this reason, I believe that the performance improvement of FCFS × HRP over batch usage of user-FB would be even greater as the degrees of speculation of the users increase.

– *Batchactive scheduling applies best when several to many speculative tasks are submitted and early task outputs are acted on while uncompleted tasks remain.*

Think time is needed to obtain time-based batchactive benefits (Figures 6.27 and 6.31). The difference in time between a user disclosing speculative work and needing a task's output, which comes from think time, motivates my 'visible' scheduling metrics (Chapter 5.2) and batchactive scheduling policies (Chapters 5.4 and 5.5). Think time is not exposed to non-speculative scheduling, and thus remains unexploited in that domain.

When all task sets have only one task, all cases provide the same time- and cost-based metrics. Disclosing task sets as small as several tasks provides good time-based improvement over interactive usage. Batch usage and batchactive usage initially improve with more tasks per task set for time-based metrics because there is think time that can be leveraged to run disclosed tasks; soon batch usage becomes unusable as its single queue is overwhelmed with speculative tasks. Over the range of task set sizes, batchactive usage of a batchactive scheduler is always best for time-based metrics and mean scaled billed resources, and batchactive scheduling's requested load is better than interactive but worse than batch usage of a non-speculative scheduler. See below for requested load commentary. (Figures 6.22, 6.23, 6.24, 6.47, 6.58, and 6.59.)

Even when all tasks in a task sets are needed (viz., the task sets were not strictly speculative), batchactive scheduling provides better time- and cost-based metrics than common practice, though not as much benefit for time-based metrics as when task sets are speculative. (Figures 6.20 and 6.21.)

- HRP *schedules disclosed tasks well,* even as well as the impractical RFCFS algorithm (Figures 6.62 and 6.63). The HRR disclosed queue subpolicy has the advantage over HRP of not penalizing users who disclose deeply but rarely request. However, its time- and cost-based performance is poorer than HRP and very similar to FCFS. I have not found an advantage to employing a subpolicy that favors the speculative tasks of users who have historically requested more work. HRR spends too much time on unneeded long-shot speculation from users who have at some point requested large tasks. (Figures 6.32, 6.33, 6.34.)

  The disclosed queue must be scheduled carefully to avoid a diminishing returns of batchactive improvement as the disclosed queue fills with tasks less likely to be requested. This is why FCFS × HRP does much better than FCFS × FCFS for time-based metrics (and marginally better for requested load) when varying the number of tasks in a task set (Figures 6.39, 6.40, and 6.47). When disclosed queues are not very deep, the simplest batchactive policy, FCFS × FCFS, provides the bulk of the performance improvements (Figure 6.45). Promising non-two-tiered batchactive schedulers which might schedule disclosed tasks better than HRP were presented in Chapter 5.4. It remains future work to study and devise practical ways to deploy them.

  The batchactive interface which separates disclosure from requests enables the benefits offered by FCFS × HRP, resulting in the better cited metrics. The proposed interface changes to support such scheduling and to encourage speculation, and whether existing interfaces can support these properties, were discussed in Chapter 5.6.

- *Service time and think time affect batchactive improvement in nearly opposite ways.* As service time increases, the time-based performance of interactive usage of a non-speculative scheduler and batchactive usage of a batchactive scheduler converge. As a limiting case, when a server is always running requested work, batchactive scheduling does not improve performance over interactive non-speculative scheduling. Likewise, the time-based performance of batch usage of a non-speculative scheduler and batchactive usage of a batchactive scheduler converge when service time decreases as there is less needed work to perform. (Figure 6.25.)

Batchactive usage of a batchactive scheduler and batch usage of a non-speculative scheduler would converge once there is enough think time for the batch configuration to execute every task from every user's current task set. Batchactive scheduling will always outperform interactive usage of a non-speculative scheduler as think time grows because the interactive configuration exposes no speculation to the system. (Figure 6.27.)

- *Batchactive improvements hold for both non-size-based (*FCFS *v.* FCFS × FCFS*) and size-based (*SRPT *v.* SRPT × FCFS*) policies to the same extent,* which is useful because size cannot always be obtained or predicted. (Figures 6.6, 6.51, 6.7, 6.52, 6.8, 6.53, 6.9, and 6.54.)

  Further, the non-size-based batchactive policy FCFS × FCFS outperforms the size-based non-speculative cases, implying that *the availability of a task size oracle will not diminish the value of batchactive scheduling.* (Figures 6.10 and 6.55.)

- *A batchactive scheduler consumes additional load for disclosed, speculative work,* and to a lesser extent, for needed tasks from users who receive outputs faster and thus request tasks faster. As the load of batchactive scheduling and interactive usage of a non-speculative scheduler approach 1, their visible response times converge. The tradeoff for batchactive schedulers improving visible response time relative to interactive usage of a non-speculative scheduler is increased load. Compared to batch usage of a non-speculative scheduler, however, the batchactive case induces little extra load while delivering significantly better visible response time. (Figures 6.10, 6.14, and 6.11.)

- Batch users on non-speculative schedulers pay for unneeded speculation, as reflected by mean scaled billed resources over 1. *By not charging users for speculative tasks, batchactive scheduling should motivate users to disclose deeply.* (Figures 6.8, 6.53, and 6.61.)

- Interactive usage of a non-speculative scheduler will earn the resource provider the least revenue because these users only request one task at time, the tasks they need to continue. When the system is not saturated, so that resources are going idle anyway, batchactive usage of a batchactive scheduler delivers significantly reduced visible response time relative to interactive usage of a non-speculative scheduler; charging only for requested tasks delivers better use of unutilized cycles against the interactive case (Figures 6.10 and 6.11). *A batchactive scheduler often provides more total billed resources over the same time period (higher requested load) compared to interactive usage of a non-speculative policy,* because, although in both situations only requested resources are charged, the batchactive case provides better visible task throughput as explained in Chapter 4.5.4. (Figures 6.14 and 6.12.)

On the other hand, batch usage of a non-speculative scheduler will earn the resource provider the most revenue because users are requesting speculative work and all executed work, whether needed or not, is charged. (In batchactive scheduling, some computing resources are consumed without being billed; these are the disclosed tasks that are never requested, as shown in Figure 6.13.) Under low to medium load, *batchactive scheduling delivers worse server revenue compared to batch usage of a non-speculative scheduler* (Figure 6.12). This is more of a boundary case than a negative result: *It is not realistic for all users to behave in a batch manner* when using a non-speculative scheduler. All users would have to be either highly confident of needing all speculative tasks immediately or have abundant resources to pay for unneeded speculative tasks. When the system approaches saturation from needed tasks, so that both schemes charge the same amount, batch usage of a non-speculative scheduler delivers worse throughput and mean visible response time. As mentioned above, latency-sensitive users will not push non-speculative scheduling into high requested loads. I suspect that *the significant value provided with batchactive scheduling with respect to time-based metrics could encourage additional users, deeper speculation, and bigger tasks, any of which would raise batchactive server revenue.* (Figures 6.14, 6.10, and 6.12.)

Ignoring the potential for users to change their behavior based on scheduling performance, in the worst case (all users behaving in a batch manner), experiments show that the resource provider could price requested resources roughly 5% more to meet the revenue of a non-speculative pricing mechanism for the task and user characteristics I evaluated.[15] (Figures 6.9, 6.44, and 6.54.) This may be a bargain for the user who would experience significantly lower visible response times, lower visible slowdowns, higher visible task throughputs, and

[15] An alternative to raising the price for requested resources is to charge a reduced amount of unneeded speculation. I dismiss this approach because it could dissuade users from disclosing speculation.

lower variances of visible response time. (Figures 6.10, 6.56, and 6.14 and Table 6.5.) In fact, batchactive users might still pay less with higher prices compared to a non-speculative pricing mechanism which charges for needless speculation. (Figures 6.8, 6.18, 6.23, and 6.28.) When interactive users are present, higher prices would entail paying more relative to the non-speculative scheduling pricing mechanism. However, the more interactive users, the less the resource provider would have to increase prices to match traditional revenue. At an extreme of all interactive users, batchactive scheduling provides better server revenue (Figures 6.9, 6.44, and 6.54). Thus, at a certain ratio of batch and interactive usage, no price change would be required and resource providers would profit more, users would pay less, and users would experience better time-based metrics.

In a deployed system, the resource provider would evaluate, given actual loads and actual user behaviors, how to set resource price to meet revenue goals, retain customers, and encourage more customers. When operating a busy resource is more expensive than an idle resource, resource price would also be set to cover losses from the uncharged load such as exhibited in Figure 6.36.

- *For a small percentage of runs, batchactive usage of a batchactive scheduler did worse compared to common practice* for mean visible response time and mean visible slowdown (performing worse than requested load was discussed in the previous point). This was apparent in the inverse cumulative improvement graphs as the areas above the curves to the left of improvement equal to 1. Some of these cases do not represent an actual performance drop but instead represent error introduced by variations in visible task throughput: different schedulers complete different numbers of needed tasks during the same simulated time. Thus metrics among two schedulers running with the same parameters are tallied from a different number of needed tasks that completed. A small subset of runs shown in Chapter 6.3.2 suggest that for the cases in which a batchactive scheduler's metric was worse, the difference was small enough to be insignificant (i.e., that the 95% confidence intervals overlap between the metric of a batchactive scheduler and to what it was being compared). The remaining cases of worse performance may result from the counterexample to the notion that a two-tiered batchactive scheduler can never perform worse than batch usage of a non-speculative scheduler (Chapter 5.5.2).

6.2.3 Graph formats

In the graphs below, simulations are identified by the scheduling policy and user behavior. Non-speculative policies consist of a single name, like ‘FCFS,’ while speculative policies are notated by the subpolicies servicing their requested and disclosed queues, like ‘FCFS × FCFS.’ The type of user is stated following the policy when a non-speculative policy is used, while batchactive users are implied when a batchactive policy is used. Batch users are labeled ‘batch’ and interactive users ‘interactive’ (often shortened to ‘inter.’). Complete examples are ‘SRPT, batch’ and ‘FCFS × HRP.’

Summarizing results report factors of *improvement* between proposed user behaviors and scheduling policies and comparison behaviors and policies. For example, if a metric is better when lower and if a comparison configuration gave 50 as the metric and a proposed configuration gave 25, then the improvement is 2. Improvement results are presented as inverse cumulative distribution graphs that show the fraction of runs in which the performance of a proposed configuration was at least a certain factor better than one or more comparison configurations. The horizontal axes are improvements and the vertical axes are the fractions of runs. For example, in Figure 6.6, the solid line intersection with the horizontal axis at 3 indicates that in about 10% of the runs, the improvement of mean visible response time for batchactive users using FCFS × FCFS compared to interactive users using FCFS was at least 3. Inverse cumulative graphs are also used to present the mean scaled billed resources of batch users on non-speculative schedulers (e.g., Figure 6.8). For those graphs, the vertical axis indicates the fraction of runs in which this metric was at least the value indicated on the horizontal axis. The simulation parameters used for these graphs cover the ranges in Table 6.1 unless otherwise noted.

Besides reporting improvement over many varying parameters, I also analyze the effect of varying parameters one at a time using bar and line graphs. For each of these graphs, the horizontal axis represents the varied parameter and the vertical axis represents the dependent metric. Each selection of parameters corresponds to a set of values, one value per user behavior and simulated scheduler. The fixed parameters for these experiments are listed in Table 6.2 unless otherwise noted. A sample graph of this format is depicted in Figure 6.10. The graph compares three configurations with respect to mean visible response time as the number of users is varied while the other parameters are fixed. Some graphs place metrics on both axes to show the relationship between them as a parameter is varied; e.g., Figure 6.15.

6.2.4 Benefits of two-tiered FCFS

The first experiment compares batch usage of FCFS, interactive usage of FCFS, and batchactive usage of FCFS × FCFS; the simplest non-speculative scheduling v. the simplest two-tiered batchactive scheduling. Task size is not needed by these schedulers.

Concerning time-based metrics, the improvement factors of mean visible response time and mean visible slowdown are shown in Figures 6.6 and 6.7, respectively. Recall that the parameters were swept through the ranges in Table 6.1 for these summarizing graphs. For half of the cases, batchactive scheduling provides better mean visible response time and mean visible slowdown than non-speculative scheduling. The mean visible slowdown improvements emphasize the differences in the mean visible response time improvements: there are more extreme cases in which batchactive scheduling does both better and worse than non-speculative scheduling. The reasons for batchactive improvements are examined in the per-parameter investigations below.

Concerning cost-based metrics, the charges for unneeded speculation incurred by batch users of a non-speculative scheduler are shown in Figure 6.8 and the improvement factors of requested load (reflecting server revenue) are shown in Figure 6.9. Again, parameters were taken from Table 6.1. Batchactive users always pay less than users submitting batches of work to non-speculative schedulers. (The mean scaled billed resources of a batchactive system is always 1.) A batchactive system often generates slightly more revenue than a non-speculative system populated by users who never submit unneeded work (viz., the interactive users). A batchactive system usually generates less revenue than a non-speculative system populated by those users who pay more by submitting batches of work, some of which will not be needed (viz., the batch users). It is unlikely that users will behave in a batch manner: it assumes that all users are highly confident of needing all speculative tasks immediately or that all users have the resources to pay for unneeded speculation. Further, these users would experience significantly worse mean visible response time at the higher levels of server revenue. Below I vary individual parameters to show the conditions under which the cost-based metrics differ most.

Note that Figures 6.6, 6.7, 6.8, and 6.9 do not include simulations of large task sets which are covered in the per-parameter investigations below. Large task sets favor batchactive scheduling, and thus these improvement factors are conservative.

For the runs covered in the above experiments, Table 6.4 reports the

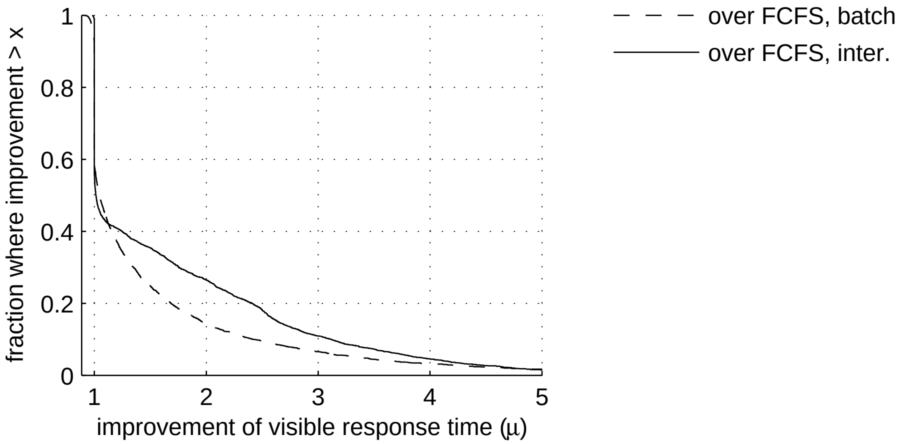


Figure 6.6: Improvement of batchactive usage of FCFS × FCFS over interactive and batch usage of FCFS for mean visible response time. A task's visible response time is the time between a user needing and receiving the task's output. The horizontal axis shows improvement factors; e.g., 2 if batchactive visible response time was half of the non-speculative visible response time for a particular set of parameters which describe user and task characteristics. Improvement values less than 1 indicate that batchactive scheduling did worse. The vertical axis indicates the fraction of the runs exploring 5,400 sets of parameters in which the improvement was *at least* as much indicated on the horizontal axis. In these graphs, batchactive performance is measured by the area under the curves for horizontal axis values greater than 1 minus area above the curves for horizontal axis values between 0 and 1. Thus, FCFS × FCFS *performs at least twice as well for about* 15% *and* 25% *of the simulated behaviors of batch* FCFS *and interactive* FCFS*,* respectively. The arithmetic mean improvement of FCFS × FCFS is 1.648 over interactive FCFS and 1.469 over batch FCFS. The geometric mean improvement of FCFS × FCFS is 1.432 over interactive FCFS and 1.308 over batch FCFS.

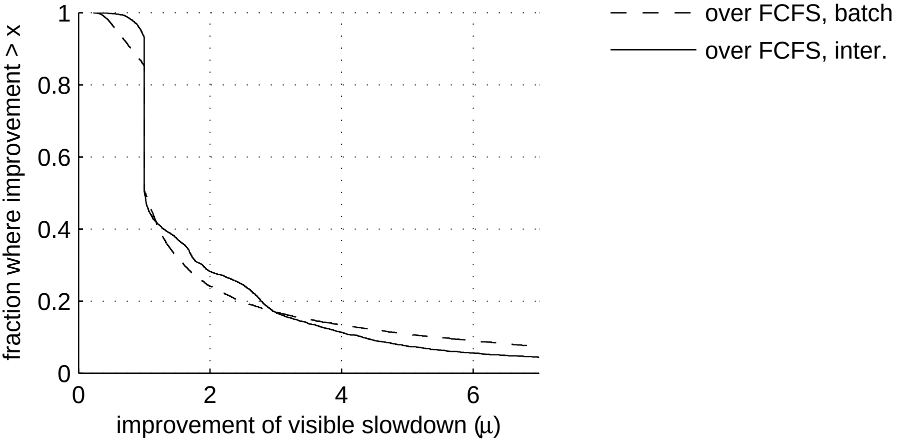


Figure 6.7: Improvement of batchactive usage of FCFS × FCFS over interactive and batch usage of FCFS for mean visible slowdown. A task's visible slowdown is its visible response time (the time a user is blocked waiting for its output) divided by the task's size. The horizontal axis shows improvement factors while the vertical axis indicates the fraction of runs in which the improvement was *at least* as much indicated on the horizontal axis. More area under the curves for horizontal axis values greater than 1 indicates better batchactive performance. Data along the line where the horizontal axis equals 1 indicates equal performance among the scheduling alternatives for particular user and task characteristics. FCFS × FCFS *performs at least twice as well for about* 25% *and* 30% *of the simulated behaviors of batch* FCFS *and interactive* FCFS, respectively. However, there are more cases where FCFS × FCFS performs worse for visible slowdown than visible response time in Figure 6.6, indicated by the area above the curves for improvement values less than 1. The arithmetic mean improvement is 2.252 over interactive FCFS and 2.892 over batch FCFS. The tail of the improvement over batch FCFS curve is heavier, resulting in its higher mean improvement.

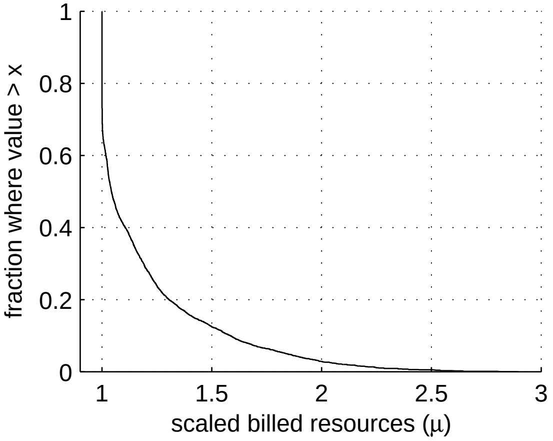


Figure 6.8: Mean scaled billed resources for batch usage of FCFS. For one user, scaled billed resources is the amount of resources charged over resources needed. Users who behave in a batch manner often pay for more resources than they need, as speculative tasks are billed and the user later determines that their outputs are unneeded. FCFS × FCFS (not shown) charges less because disclosed tasks are not charged according to the batchactive pricing mechanism. The horizontal axis shows mean scaled billed resources while the vertical axis indicates the fraction of runs in which the mean scaled billed resources was at least as much indicated on the horizontal axis. More area under the curve indicates higher costs for batch users. *The average batch user using* FCFS *pays at least* 30% *more than necessary for* 20% *of the runs.* Over all 5, 400 runs, the average mean scaled billed resources is 1.182.

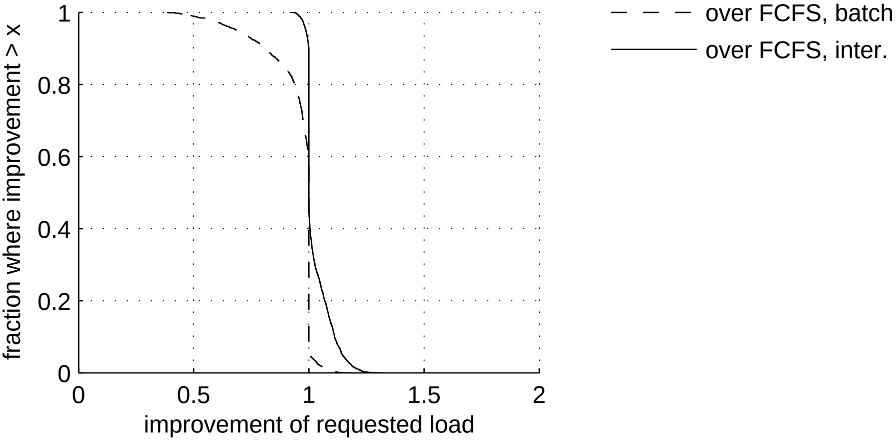


Figure 6.9: Improvement of batchactive usage of FCFS × FCFS over interactive and batch usage of FCFS for requested load. Requested load reflects the fraction of billable resource time. Requested load is equivalent to load for interactive and batch usage because non-speculative scheduling does not distinguish between disclosed and requested tasks. Further, for batchactive usage, requested load reflects only the resources used by tasks whose outputs users have asked for. The horizontal axis shows improvement factors of FCFS × FCFS while the vertical axis shows the fraction of the runs in which the improvement was *at least* as much indicated on the horizontal axis. Better batchactive performance is reflected by the area under the curves for improvement values greater than 1, while worse batchactive performance is reflected by the area above the curves for improvement values less than 1. Data along the line where improvement equals 1 reflects equal performance for some user and task characteristics. FCFS × FCFS *provides better requested load than interactive* FCFS *and worse requested load than batch* FCFS. Doing worse against batch FCFS is expected and was explained in Chapter 6.2.2. The arithmetic mean improvement is 1.029 over interactive FCFS and 0.952 over batch FCFS. (A mean improvement factor less than 1 is overall worse performance.)

FCFS, batch	FCFS, interactive	FCFS × FCFS
791.610 (99.690)	0.000 (0.000)	197.464 (20.538)

Table 6.4: The number of deadlines met among batch usage of FCFS, interactive usage of FCFS, and batchactive usage of FCFS × FCFS. A deadline for a task is met if it has executed before the user needs its output. Therefore, a higher deadlines met is better. The mean of all runs for each scheduler is presented. The 95% confidence interval of each mean is the mean plus and minus the value in parenthesis. FCFS × FCFS *performs better than interactive* FCFS *and worse than batch* FCFS. Interactive FCFS never runs a task before needed (its 'deadline'). Batch FCFS continually aids tasks until they complete.

FCFS, batch	FCFS, interactive	FCFS × FCFS
20,182.380 (755.277)	4,542.007 (101.280)	4,407.839 (103.401)

Table 6.5: The standard deviation of visible response time among batch usage of FCFS, interactive usage of FCFS, and batchactive usage of FCFS × FCFS. This metric reflects how different visible response times are. Users dislike variability; lower standard deviations are better. The mean of all runs for each scheduler is presented. The 95% confidence interval of each mean is the mean plus and minus the value in parenthesis. *Batchactive scheduling improves the variance of visible response time.*

number of deadlines met. Interactive FCFS cannot meet deadlines because tasks are requested only after needed. Batch FCFS surprisingly meets more deadlines than FCFS × FCFS. This is because FCFS × FCFS shifts attention to requested tasks in an attempt to minimize visible response time while batch FCFS continually aids tasks until they complete.

Table 6.5 confirms that the variance of visible response time for the runs covered in the experiments above does not become worse for FCFS × FCFS batchactive scheduling. In fact, it improves under batchactive scheduling as tasks that are not known to be needed are deferred when known-needed work is present and lower visible response times are achieved.

I now show the effects of varying individual parameters. I start with varying the number of users while holding other parameters constant at the values in Table 6.2.

Figure 6.10 shows how the number of users affects mean visible response time. Nearly always, batchactive FCFS × FCFS performs best, exhibiting adaptability. Batchactive FCFS × FCFS is better than batch FCFS under many users because requested tasks never wait for speculative tasks; it is better than interactive FCFS under few users because it fills idle time with speculative tasks. At the busiest part, interactive FCFS and batchactive FCFS × FCFS begin to converge because the requested task queue of the batchactive scheduler is never empty (Figure 6.12).

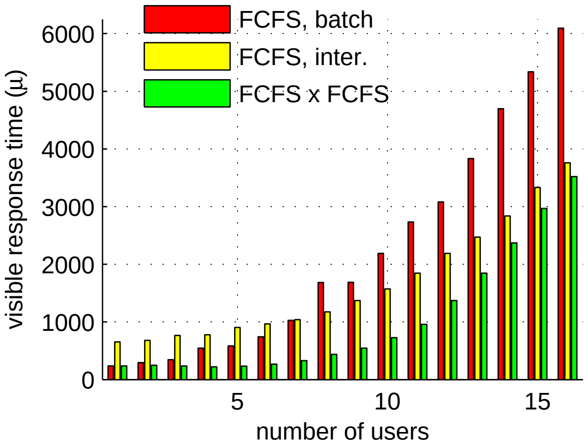


Figure 6.10: The effect of the number of users on batchactive usage of FCFS × FCFS, interactive usage of FCFS, and batch usage of FCFS for mean visible response time. Each set of three scheduling configurations was run with a different number of users simultaneously competing for the shared resource while other user and task characteristics were held constant. Visible response time for a task is the time a user was blocked on its output. The different configurations are represented by bars of different shades, with the right-most bar being the batchactive configuration. Since lower visible response time is better, bars of less height indicate better performance. With few users, batch FCFS is better than interactive FCFS; with many, interactive FCFS is better than batch FCFS. FCFS × FCFS *adapts and always performs best.*

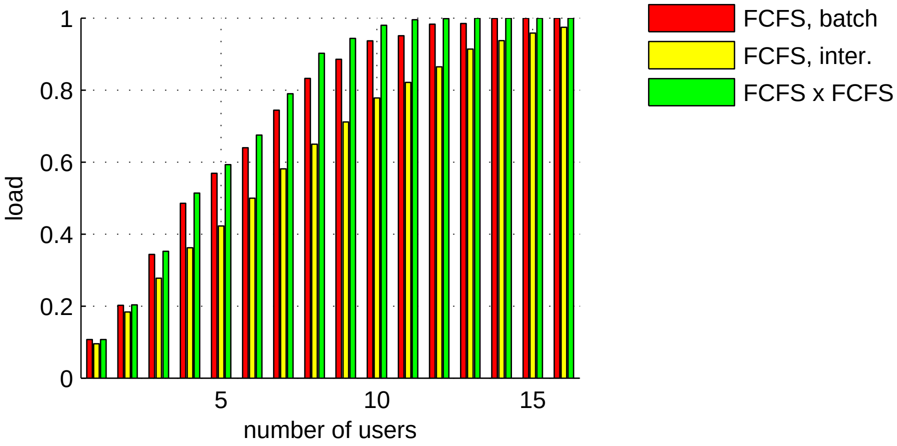


Figure 6.11: The effect of the number of users on batchactive usage of FCFS × FCFS, interactive usage of FCFS, and batch usage of FCFS for load. Load is the fraction of time that the resource was busy. High load is not necessarily good or bad; it is presented to facilitate the understanding of server capacity. Instead, a server wishes to maximize *requested* load, which is different for the batchactive case, as shown in Figure 6.12. FCFS × FCFS *uses more load than interactive and batch* FCFS. This additional load is used for disclosed tasks and for requested tasks from users who are receiving greater throughput and thus request tasks more quickly. Batch FCFS induces more load than interactive FCFS, because, in the latter case, only one task per user is in the system at any time.

Comparing Figure 6.10 with Figure 6.11 shows how (total) load is affected by varying the number of users. The load under batch FCFS and FCFS × FCFS are similar. As their loads increase, FCFS × FCFS provides better visible response time than batch FCFS because FCFS × FCFS favors requested tasks. FCFS × FCFS provides better visible response time than interactive FCFS because FCFS × FCFS pipelines disclosed tasks with user think time. When FCFS × FCFS's load is higher than interactive FCFS, it still performs better than interactive FCFS. This additional load reflects tasks that have been or may be requested. As the loads of FCFS × FCFS and interactive FCFS approach 1, their visible response times converge.

Not all of this additional load, which consists of tasks that have been or may be requested, is charged. Resource provider revenue is instead a function of requested load, shown in Figure 6.12 as the number of users is varied. FCFS × FCFS provides better requested load than interactive FCFS because, by providing better mean visible response time, users submit needed work more quickly. FCFS × FCFS is roughly 10% better than interactive FCFS un-

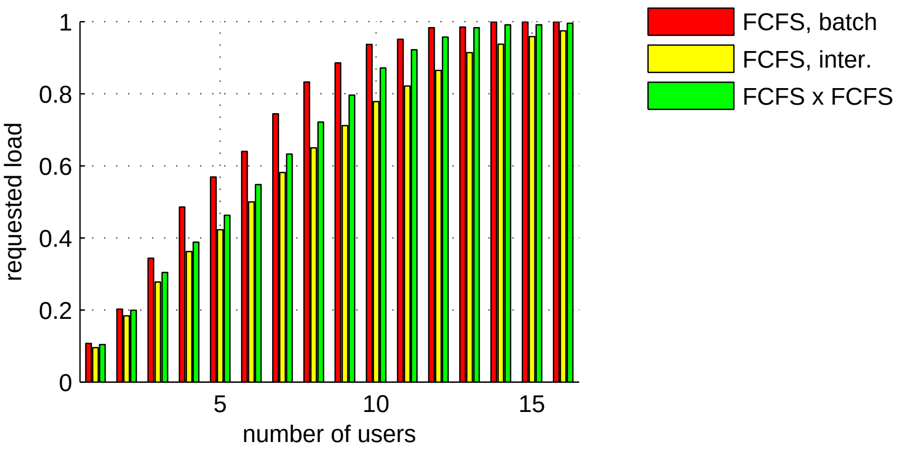


Figure 6.12: The effect of the number of users on batchactive usage of FCFS × FCFS, interactive usage of FCFS, and batch usage of FCFS for requested load. Requested load is the load that is charged. Since higher requested load is better for the server, bars of greater height indicate better server utility. Compared to interactive FCFS, FCFS × FCFS provides better requested load. FCFS × FCFS *can only match batch* FCFS*'s request load at a high number of users.*

til saturated. FCFS × FCFS cannot meet the requested load of batch FCFS, in which users request their entire computational plans and are billed for whatever fraction of these plans had executed before being needed or canceled. The uncharged load of the FCFS × FCFS case is shown in Figure 6.13. Batch FCFS assumes a willingness for users to pay for potentially unneeded speculation, even in the face of dismal mean visible response time. This issue was discussed in Chapter 6.2.2.

Figure 6.14 shows how the number of users affects visible task throughput. Batchactive FCFS × FCFS achieves a higher visible task throughput while providing the better mean visible response time as shown in Figure 6.10.

Almost always, batchactive scheduling provides better mean visible response time and better visible throughput. It is useful to see for specific runs the relation between these better quantities. I show that a higher number of needed tasks get through the system at an acceptable mean visible response time by graphing visible throughput on the horizontal axis and mean visible response time on the vertical axis in Figure 6.15. These two dependent variables are a function of changing the number of users (not shown) while holding other parameters constant. This graph shows that batchactive scheduling's combination of mean visible response time and visible through-

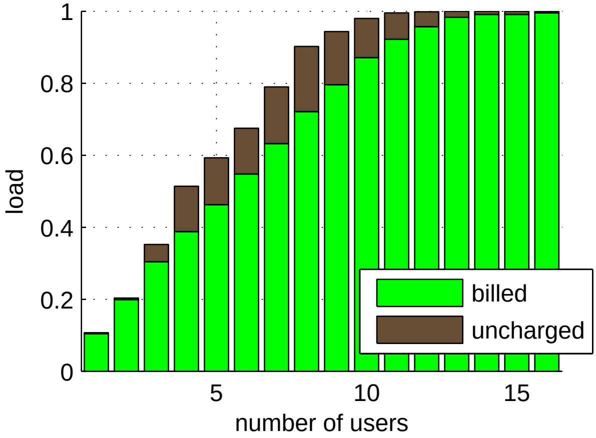


Figure 6.13: The effect of the number of users on batchactive usage of FCFS × FCFS for the requested (billed, charged) and uncharged load. The uncharged load is used server time that is not charged to any user under the batchactive pricing mechanism. This figure provides the uncharged load information for the batchactive case of Figure 6.12. *At medium load, the server loses potential revenue from instituting the batchactive pricing mechanism.* This revenue loss is unlikely in practice, as discussed in Chapter 6.2.2.

put is always better (sometimes much better) than interactive and batch usage of non-speculative scheduling.

Almost always, batchactive scheduling provides better mean visible response time. Often, batchactive scheduling provides worse requested load (server revenue) compared to batch usage of a non-speculative system. The following graph shows the relation of these quantities for specific runs by varying the number of users while holding other parameters constant. Although there are more cases in which batch usage of FCFS provides optimal (1) requested load, this comes at a price of mean visible response time. I show that there is more billable time at an acceptable mean visible response time by graphing requested load on the horizontal axis and mean visible response time on the vertical axis in Figure 6.16.

Together, Figures 6.15 and 6.16 show that the threshold of overload is better with a batchactive scheduler; i.e., the non-speculative schedulers' visible mean visible response times increase faster at lower visible throughputs and requested loads.

Now I vary the upper bound of the task set change probability, holding the other parameters constant.

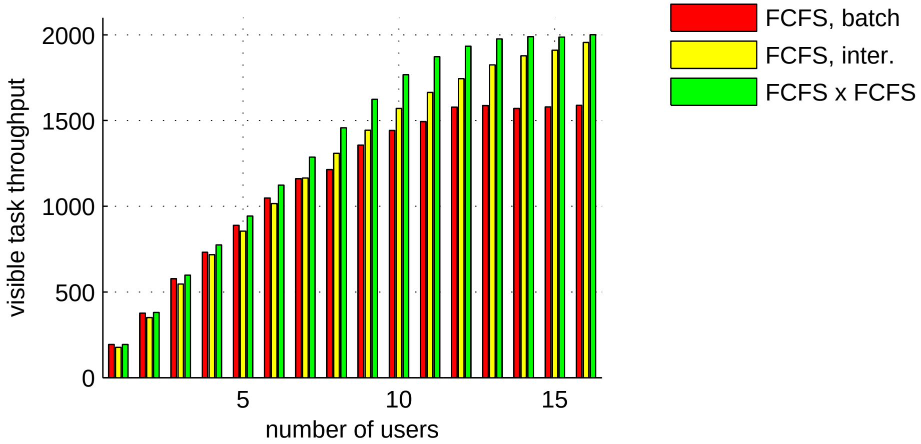


Figure 6.14: The effect of the number of users on batchactive usage of FCFS × FCFS, interactive usage of FCFS, and batch usage of FCFS for visible task throughput. Visible task throughput is the number of needed tasks that have executed; i.e., the number of tasks whose outputs were needed which were delivered to their submitting users. Higher visible task throughput is better. Visible throughput is shown to illustrate that it does not worsen as metrics such as mean visible response time improve. Improving visible throughput is not a goal in itself. A server is concerned with billable resources, which includes resources used by needed and unneeded tasks (and fractions thereof) under the non-speculative pricing mechanism. FCFS × FCFS *provides the best visible task throughput.*

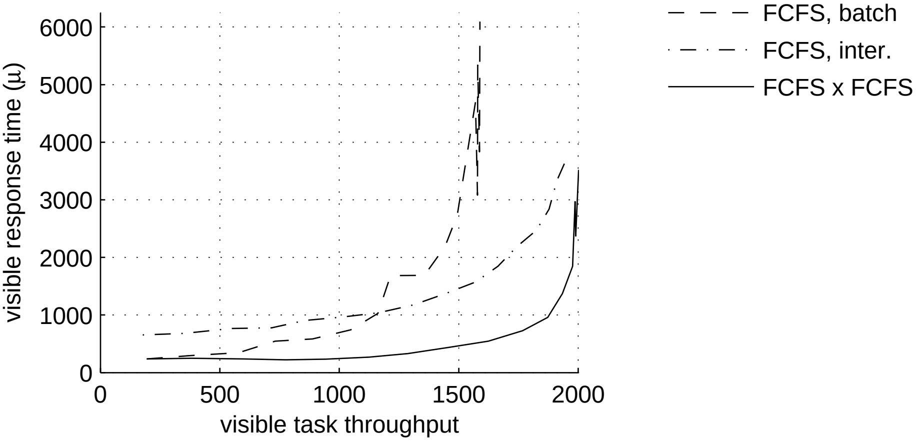


Figure 6.15: The relationship on batchactive usage of FCFS × FCFS, interactive usage of FCFS, and batch usage of FCFS between visible throughput and mean visible response time as the number of users was varied from 1 to 16. Both axes are dependent axes; the horizontal axis indicates visible task throughput while the vertical axis indicates mean visible response time. The number of users was varied while other parameters were held constant. The more the batchactive curve is below and to the right of the other curves, the better its performance, as that indicates better visible throughput at better mean visible response time. FCFS × FCFS *always provides better pairs of visible throughput and mean visible response time.* At a visible throughput of 1,500, the visible response time of FCFS × FCFS is over 3 times better than interactive FCFS and over 6 times better than batch FCFS. Note that this particular improvement was achieved with a different number of users for the different scheduling configurations. That is, for any visible task throughput or any visible response time, it is likely that a different number of users resulted in that performance for the different scheduling configurations. The setting of this independent variable may be found by examining Figures 6.10 and 6.14 for number of users v. mean visible response time and v. visible throughput, respectively.

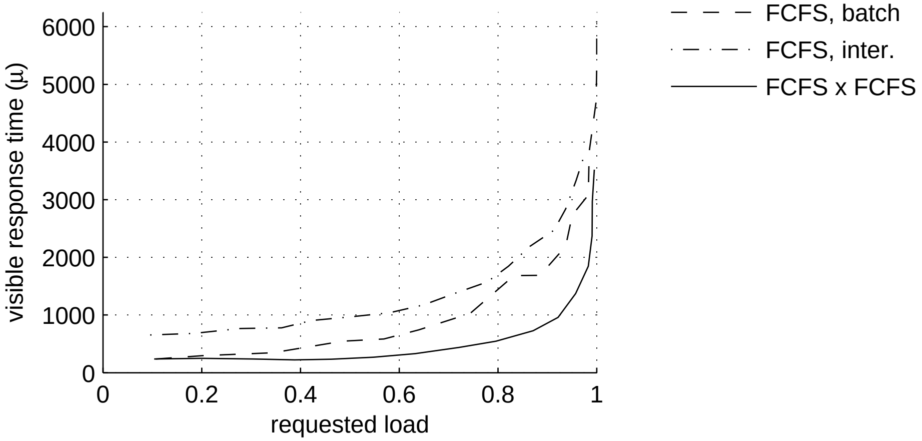


Figure 6.16: The relationship on batchactive usage of FCFS × FCFS, interactive usage of FCFS, and batch usage of FCFS between requested load and visible response time as the number of users was varied from 1 to 16. Both axes are dependent axes; the horizontal axis indicates requested load and the vertical axis represents mean visible response time. The number of users was varied while other parameters were held constant. The more the batchactive curve is below and to the right of the other curves, the better its performance, as that indicates better requested load at better mean visible response time. FCFS × FCFS *always provides better pairs of requested load and mean visible response time.* The setting of the number of users independent variable, for any achieved requested load or mean visible response time, may be found by examining Figures 6.10 and 6.12.

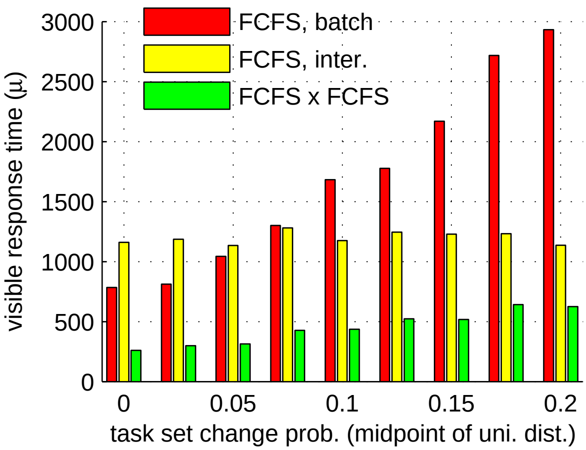


Figure 6.17: The effect of the task set change probability on batchactive usage of FCFS × FCFS, interactive usage of FCFS, and batch usage of FCFS for visible response time. Each set of three configurations was run with a different upper bound on the uniform probability that determined the likelihood that users would cancel unviewed task outputs after thinking about received outputs. Other parameters were held constant. The probability is a uniform random variable and shown on the horizontal axis is the average of its lower (always 0) and upper bounds. *Interactive* FCFS *is unaffected while batchactive* FCFS × FCFS *is affected less than batch* FCFS *and outperforms both.*

In Figure 6.17, I vary the upper bound of the task set change probability from 0.0–0.4 and plot mean visible response time. This parameter does not affect interactive FCFS as these users do not submit disclosed tasks. There is a greater dependence on this parameter with batch FCFS compared to FCFS × FCFS because FCFS × FCFS avoids speculative tasks when requested work exists.

I varied the task set change probability to show its effect on mean scaled billed resources in Figure 6.18. Batch users waste increasing money on unneeded speculation as their task sets become more speculative. This extra money is spent as mean visible response times worsen as shown in Figure 6.17.

I varied the task set change probability to show its effect on requested load in Figure 6.19. As batch users waste more money (Figure 6.18), the resource provider earns more revenue. Requested load is not a function of how often users cancel task sets for interactive use of FCFS and batchactive use of FCFS × FCFS. Requested load is higher for the batchactive case relative to interactive FCFS because the better mean visible response times achieved

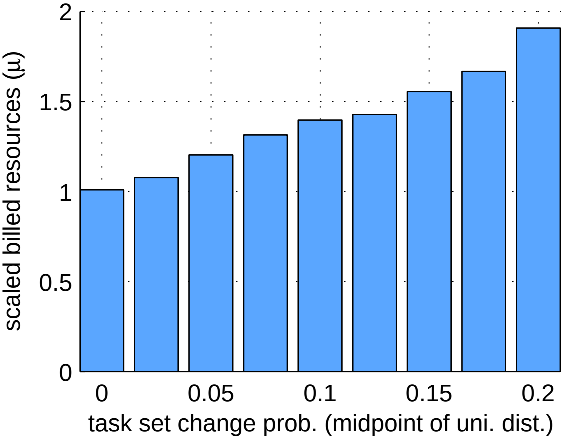


Figure 6.18: The effect of the task set change probability on batch usage of FCFS for mean scaled billed resources. *As task sets become increasingly speculative, batch users spend increasingly needlessly.*

by the former (Figure 6.17) cause users to submit needed work more quickly.

I show that there is still benefit to batchactive scheduling when users plan ahead task sets in which there is no speculation; i.e., when all tasks in one's task set will eventually be requested and never canceled. I do this by fixing the upper bound of the task set change probability to 0. Figure 6.20 shows that the benefit for mean visible response time exists but is not as great as when there is speculation (Figure 6.10), as the number of users is varied while other parameters are held constant. The mean scaled billed resources for batch FCFS (not shown) is always 1 since no disclosed work is wasted.

I repeat this non-speculative experiment but instead show requested load. The requested load shows an interesting phenomenon in Figure 6.21 as the number of users is varied while other parameters are held constant: requested load for the FCFS × FCFS case is sometimes higher than batch FCFS. While requested load was frequently higher than the interactive case because users receive outputs faster and submit more needed work faster (Chapter 4.5.4), this phenomenon also occurs against the batch case because here all disclosed work is eventually determined by the simulated users to be needed.

I now vary the upper bound of the number of tasks per task set while holding other parameters constant.

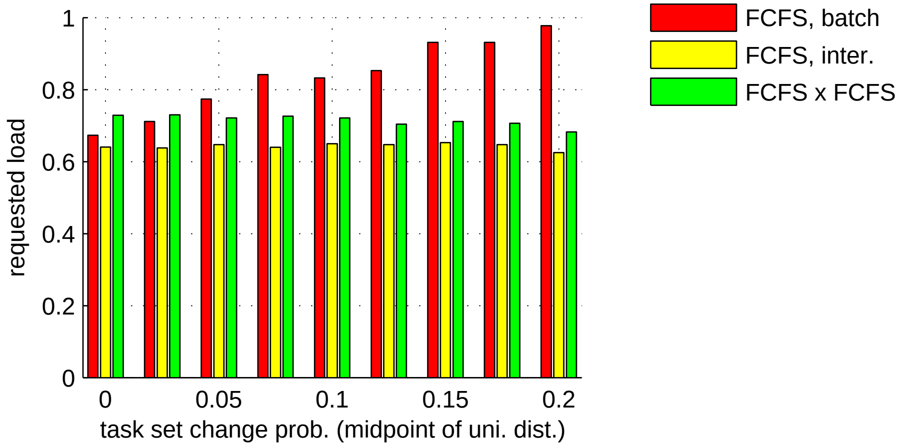


Figure 6.19: The effect of the task set change probability on batchactive usage of FCFS × FCFS, interactive usage of FCFS, and batch usage of FCFS for requested load. *The task set change probability affects only the batch case: a greater chance of cancelation increases server revenue.* But while FCFS batch gets more billed work accomplished, it has very poor visible response time (Figure 6.17).

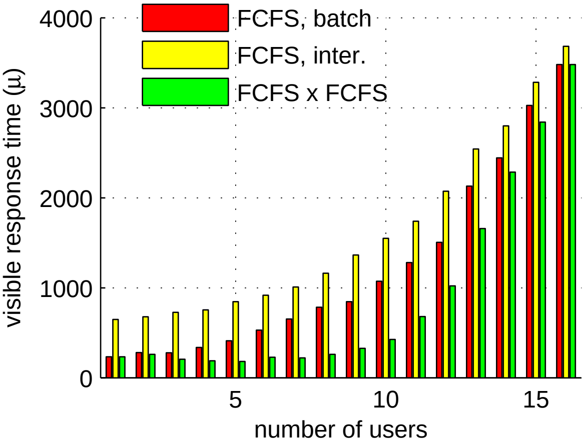


Figure 6.20: The effect of the number of users on batchactive usage of FCFS × FCFS, interactive usage of FCFS, and batch usage of FCFS for mean visible response time when all work is needed. Lower bars indicate better performance as the number of users was varied while other parameters were held constant among each scheduling configuration. *There is still benefit to batchactive scheduling when task sets are not speculative.*

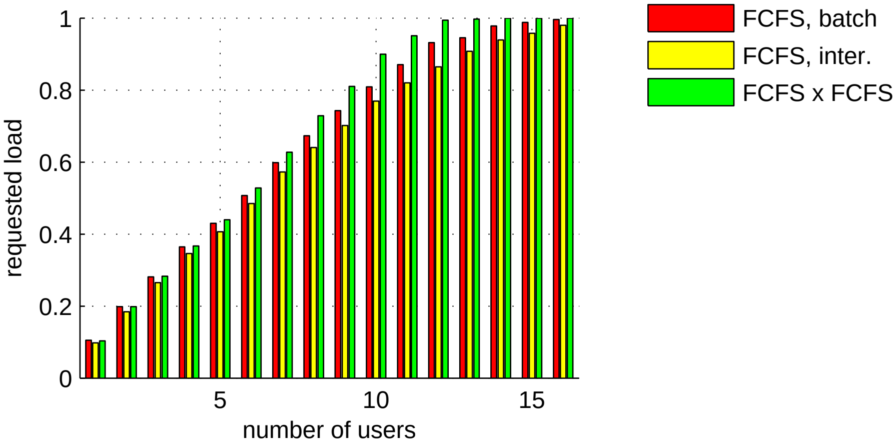


Figure 6.21: The effect of the number of users on batchactive usage of FCFS × FCFS, interactive usage of FCFS, and batch usage of FCFS for requested load when all work is needed. *Batchactive requested load consistently beats even the batch case when tasks are never canceled.*

I show large task sets reflecting users searching high-dimensional spaces. These runs were not included in the summarizing inverse cumulative graphs above (Figures 6.6, 6.7, 6.8, and 6.9). I varied the upper bound of the number of tasks per task set uniform distribution from 1–$1,024$ in multiples of 2. Its effect on mean visible response time is shown in Figure 6.22. When all task sets have only one task, all cases provide the same mean visible response time. Disclosing task sets as small as several tasks — easily realizable by users performing exploratory searches — provides good improvement over interactive FCFS. Batch FCFS and FCFS × FCFS initially improve with more tasks per task set because there is think time that can be leveraged to run disclosed tasks. Soon batch FCFS becomes unusable as its single queue is overwhelmed with speculative tasks. Interactive FCFS is immune to the number of tasks per task set because these users will have submitted at most one task from their task sets. Batchactive FCFS × FCFS is always best. Compared to itself, its performance worsens as the number of tasks per task set increases because there is more competition among speculative tasks, some of which run but are never needed.

I varied the number of tasks per task set to show the effect on mean scaled billed resources in Figure 6.23. As task set sizes increase, batch users pay greatly for unneeded speculation. This extra cost does not help their mean visible response times, as shown in Figure 6.22.

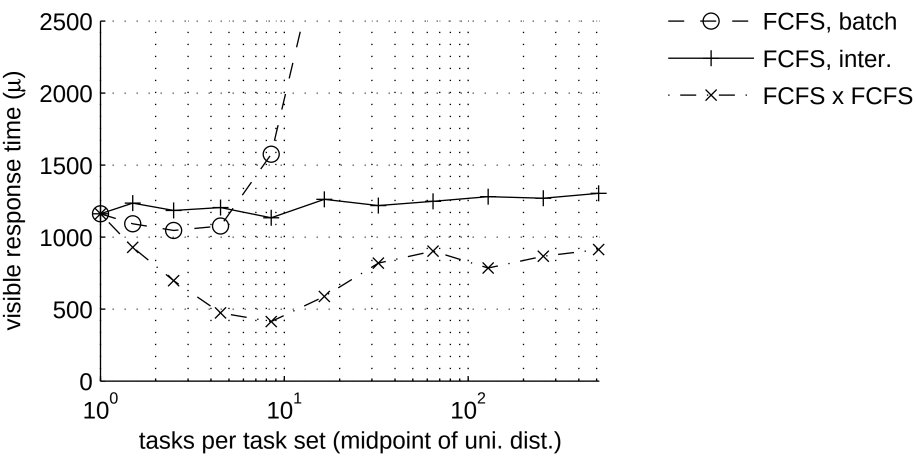


Figure 6.22: The effect of the number of tasks per task set on batchactive usage of FCFS × FCFS, interactive usage of FCFS, and batch usage of FCFS for mean visible response time. Each set of three configurations was run with a different upper bound on the uniform distribution determining the number of tasks per task set for each user; the lower bound was always held at 1. Other parameters were held constant. The number of tasks per task set is a uniform random variable and shown on the horizontal axis is the average of its lower and upper bounds. The saddle point for FCFS × FCFS at the horizontal axis value of 128 is within confidence intervals (not shown). FCFS × FCFS *always wins and batch* FCFS *quickly becomes unusable.* At a number of tasks per task set with an upper bound of 64, the mean visible response time of the batch case is $7,270$. (This graph is log-linear.)

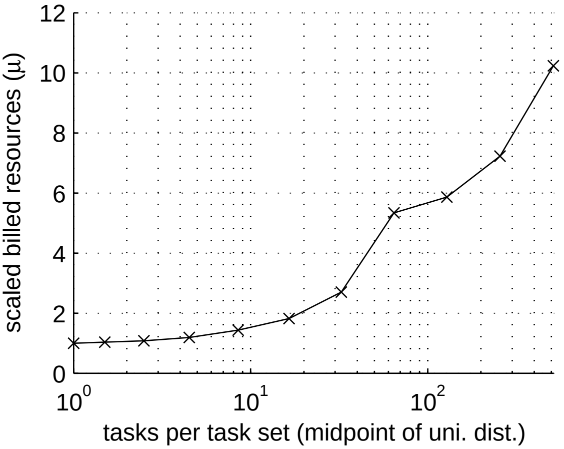


Figure 6.23: The effect of the number of tasks per task set on batch usage of FCFS for mean scaled billed resources. *As task sets become larger, batch users waste more money on unneeded speculation.*

I varied the number of tasks per task set to show the effect on requested load in Figure 6.24. With every task set consisting of one needed task, all configurations result in the same revenue. Maximum revenue for the batch case, which occurs when the upper bound on the number of tasks per task set is at least 60, results in unacceptable mean visible response time and mean scaled billed resources (Figures 6.22 and 6.23). Interactive and batchactive usage are most different when the upper bound of the number of tasks per task set is between 12 and 18, the points at which the difference of the mean visible response times between the two configurations is greatest (Figure 6.22).

Now I vary mean service time while holding other parameters constant.

Figure 6.25 shows how mean service time affects mean visible response time. As mean service time increases, the performance of interactive FCFS and batchactive FCFS × FCFS converge. As a limiting case, when a server is always running requested work, FCFS × FCFS does not improve performance over interactive FCFS.

There is no significant effect of mean service time on mean scaled billed resources, thus no data is presented.

I varied service time to show its effect on requested load in Figure 6.26. At a mean service time of approximately $1,200\,\mathrm{s}$ and above, batchactive scheduling provides equal server revenue as batch usage of FCFS while providing much better mean visible response time as shown in Figure 6.25.

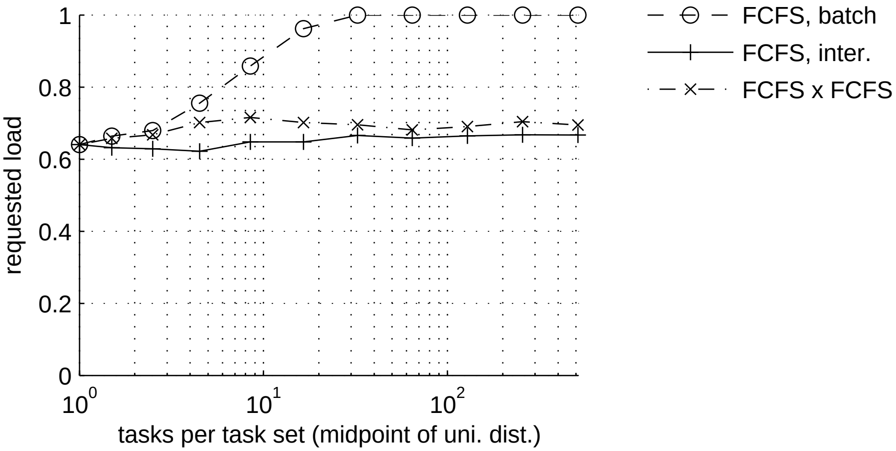


Figure 6.24: The effect of the number of tasks per task set on batchactive usage of FCFS × FCFS, interactive usage of FCFS, and batch usage of FCFS for requested (charged) load. A sweet spot shows an improvement of server revenue for the batchactive case over the interactive case. *Batch usage of non-speculative scheduling produces the best revenue at the expense of unacceptable mean visible response time* (Figure 6.22).

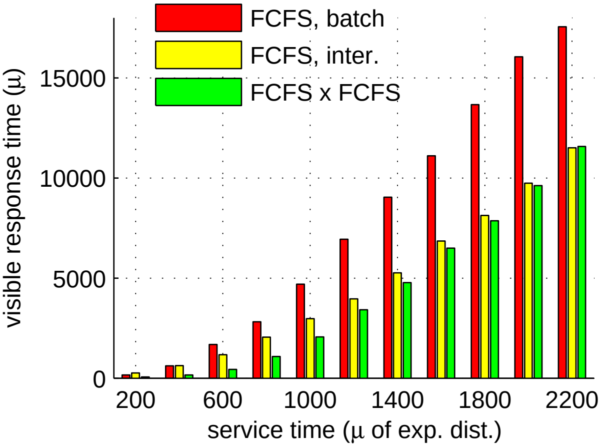


Figure 6.25: The effect of service time on batchactive usage of FCFS × FCFS, interactive usage of FCFS, and batch usage of FCFS for mean visible response time. Each set of three configurations was run with a different mean for the exponential distribution determining task service time. Other parameters were held constant. *At the low end, there is more opportunity for batchactive scheduling to improve performance.*

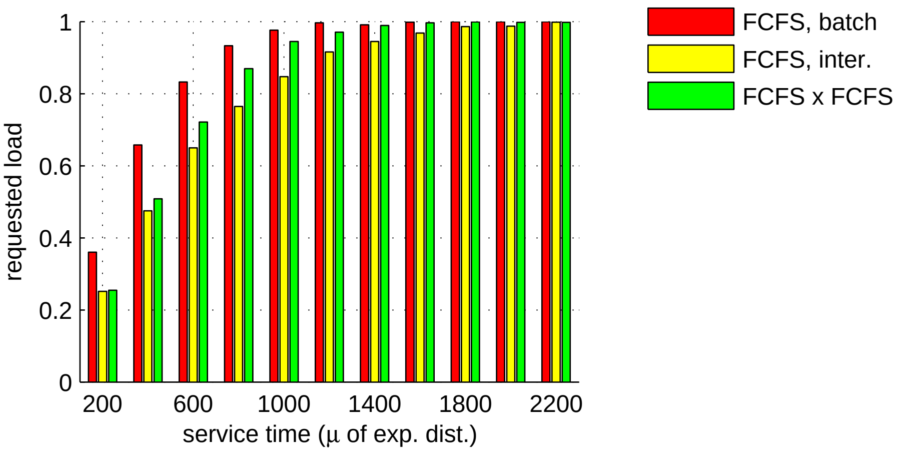


Figure 6.26: The effect of service time on batchactive usage of FCFS × FCFS, interactive usage of FCFS, and batch usage of FCFS for requested (charged) load. *When mean service time is sufficiently high, batchactive and batch usage achieve equal server revenue.*

Finally, I vary think time while holding the other parameters constant.

Think time works inversely to service time: the more think time, the more opportunity for batchactive scheduling to reduce visible response time, as shown in Figure 6.27. All three configurations provide the same performance with little think time and improve with more think time. However, FCFS × FCFS outperforms both batch and interactive FCFS with more think time in the range shown. Eventually FCFS × FCFS and batch FCFS would converge once there is enough think time for batch FCFS to execute every task from every user's current task set. FCFS × FCFS will always outperform interactive FCFS as mean think time increases because interactive FCFS exposes no speculation to the system.

I varied think time to show its effect on mean scaled billed resources in Figure 6.28. The more think time, the more likely the server will get ahead of the user, executing speculative tasks that will turn out to not be needed, resulting in charging for unneeded speculation.

I varied think time to show its effect on requested load in Figure 6.29. With little think time, server revenue is maximum across configurations. As think time increases, server revenue decreases: there are times in which users are thinking and not needing work. Batch revenue is highest because speculative work is charged. Interactive revenue is lowest because speculative work is never submitted. Batchactive revenue is between the two: unneeded spec-

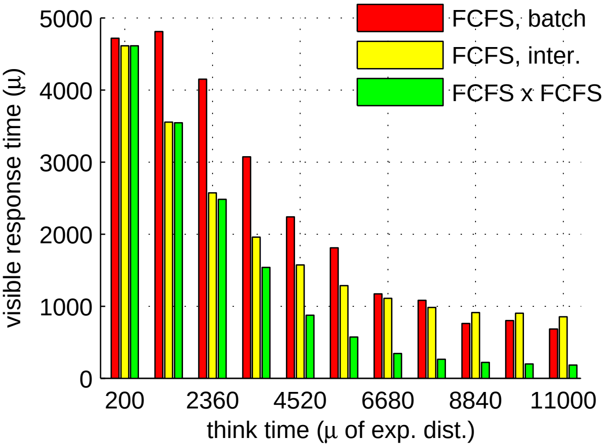


Figure 6.27: The effect of think time on batchactive usage of FCFS × FCFS, interactive usage of FCFS, and batch usage of FCFS for mean visible response time. Each set of three configurations was run with a different mean for the exponential distribution determining user think time. *At the high end, there is more opportunity for batchactive scheduling to improve performance.*

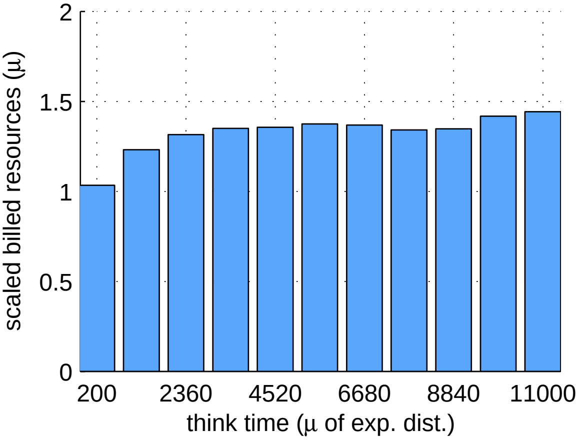


Figure 6.28: The effect of think time on batch usage of FCFS for mean scaled billed resources. *Greater think time results in more wasted money on unneeded speculation.*

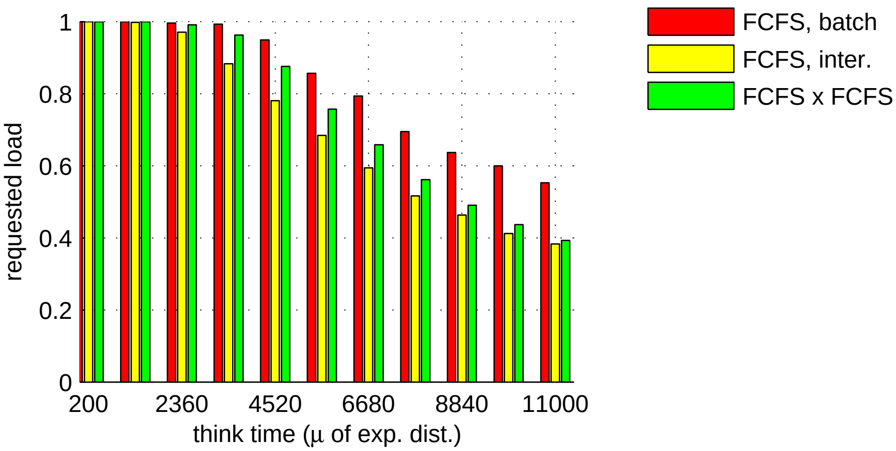


Figure 6.29: The effect of think time on batchactive usage of FCFS × FCFS, interactive usage of FCFS, and batch usage of FCFS for requested load. Requested load is proportional to server revenue. *Greater think time negatively affects server revenue.*

ulation is not charged, but since users receive outputs faster, they request needed work more quickly.

I show that the ratio of user think time to task service time is correlated to visible response time in Figure 6.30. A wide range of mean think time (200–11, 000 s in 1, 080 s increments) and mean service time (200–2, 000 s in 200 s increments) were used while other parameters were held constant. While a trend is evident, with batchactive visible response time always the lowest (best), this graph shows that performance is not a simple function of the ratio of think time and service time. When one value is substantially different from the other, it dominates in ways hard to appreciate from the ratio. From Figures 6.25 and 6.27, it is evident that changes in service time have a greater effect than changes in think time with respect to absolute visible response time, and that changes in think time have a greater effect in the relative difference between schedulers.

I confirm that there is no benefit to batchactive scheduling when users do not exhibit any think time, as discussed in Chapter 2.3.1. All three configurations result in the same performance as shown in Figure 6.31. (Further, but not shown, mean scaled billed resources is always 1 for batch FCFS, and requested load for all three configurations is 1.)

The number of scheduling decisions is shown in Table 6.6. Two-tiered FCFS-based batchactive scheduling does not cause more scheduling decisions to be made than non-speculative scheduling.

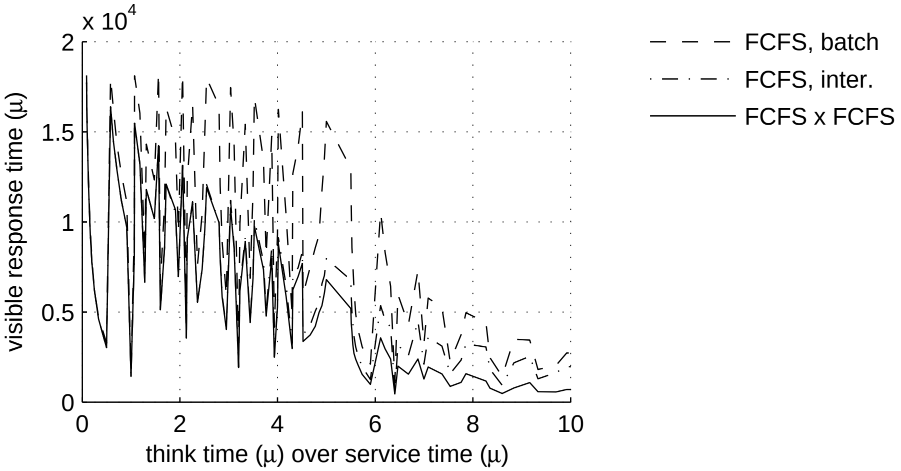


Figure 6.30: The effect of mean think time over mean service time on batchactive usage of FCFS × FCFS, interactive usage of FCFS, and batch usage of FCFS for mean visible response time. The horizontal axis indicates ratios of the means used in the exponential distributions to generate think and service times. Other parameters were held constant. Lower mean visible response time is better, so lower curves indicate better performance. The chaotic nature of this data is explained as follows: Two values visually next to each other might be comprised of very different service and think times, such as 11, 000:2, 000 (5.5) and 5, 600:1, 000 (5.6). These selections will exhibit very different mean visible response times because the effect of one parameter (service time) dominates the effect of the other (think time) as shown in Figures 6.25 and 6.27. Still, *at every point batchactive* FCFS × FCFS *is better.*

	FCFS, batch		FCFS, interactive		FCFS × FCFS	
count	2,059.7	(23.9)	2,657.6	(19.6)	2,543.3	(40.7)
scaled count	1.616	(0.045)	2.001	(0.000)	1.695	(0.032)

Table 6.6: Total number (averaged over 35 runs started with different random seeds) of scheduling decisions over two weeks of simulated time. The scaled count is the count divided by the number of finished (needed) tasks. The 95% confidence interval of each mean is the mean plus and minus the value in parenthesis. *The number of scheduling decisions for* FCFS × FCFS *is between that for interactive and batch usage of* FCFS*.* The time overhead of these decisions is presented in Table 7.1.

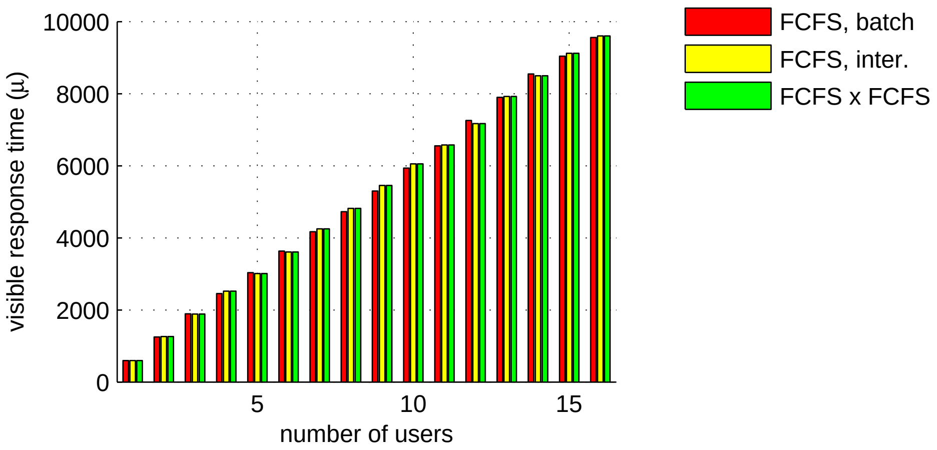


Figure 6.31: The effect of the number of users on batchactive usage of FCFS × FCFS, interactive usage of FCFS, and batch usage of FCFS for mean visible response time when think time is removed. The number of users was varied while other parameters were held constant. For each scheduling configuration, performance is the same within error, showing that *think time is necessary for batchactive scheduling benefits.*

This section covered the principal benefits of batchactive scheduling for the most-easily deployable two-tiered batchactive scheduler which does not rely on predictions of user behavior or knowledge of task size. Except requested load relative to batch users, every time- and cost-based metric improved over a wide range of scenarios.

6.2.5 Determining a better disclosed queue scheduler

Here I explore the performance of different task size-agnostic disclosed queue subpolicies. I compare the batchactive scheduler FCFS × FCFS from the previous section (Chapter 6.2.4) against FCFS × HRP and FCFS × HRR. Both HRP and HRR are novel disclosed queue subpolicies (Chapter 5.5.1). Highest-request-probability (HRP) favors disclosed tasks from users who have historically requested a greater percentage of their speculative tasks. Highest-requested-resources (HRR) favors disclosed tasks from users who have requested more resources since entering the system.[16]

Comparisons involving the mean scaled billed resources metric are not shown because this metric is always optimal (1) under the batchactive pricing mechanism introduced in Chapter 5.1.

[16] All users enter the system at the start of the simulation (Chapter 6.1.1).

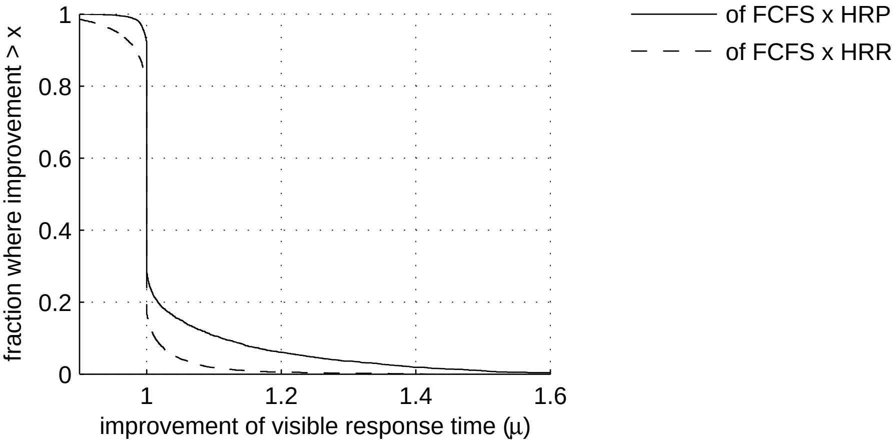


Figure 6.32: Improvement of FCFS × HRP and FCFS × HRR over FCFS × FCFS for mean visible response time. The horizontal axis indicates improvement values while the vertical axis indicates the fraction of the sets of parameters under test in which the improvement was *at least* as much indicated on the horizontal axis. Area under the curves for improvements greater than 1 indicate that the proposed disclosed queue subpolicies did better than FCFS × FCFS. Area above the curves for improvements less than 1 indicate that FCFS × FCFS did better. Data along improvement equal to 1 indicate equal performance. FCFS × HRP *provides the best mean visible response time.* The arithmetic mean improvement of FCFS × HRP is 1.033 and of FCFS × HRR is 1.001, indicating little change over the range of task and user characteristics tested. Below I vary individual parameters to show the conditions under which disclosed queue subpolicies have the most effect.

The improvement factors of the time-based metric mean visible response time is shown in Figure 6.32. FCFS × HRP provides better mean visible response times than FCFS × FCFS. Surprisingly, the performance of FCFS × HRR is, on average, nearly the same as FCFS × FCFS, as shown by the nearly equal areas below and to the right and above and to the left of improvement equal to 1. A breakdown of the cases where better and worse performance occurs is presented below when parameters are varied individually.

The cost-based metric, requested load, is shown in Figure 6.33. Both FCFS × HRP and FCFS × HRR provide better server revenue than FCFS × FCFS. At the outset, I had expected HRR to provide better server revenue than HRP because it favors users who had requested more work historically. Instead HRR spends too much time on unneeded long-shot speculation from users who have early on requested large tasks. More detail is found in the per-parameter investigations below.

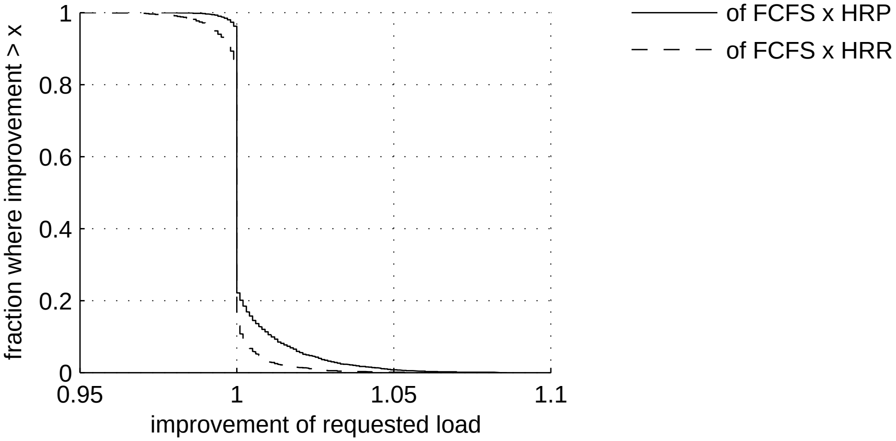


Figure 6.33: Improvement of FCFS × HRP and FCFS × HRR over FCFS × FCFS for requested (charged) load. FCFS × HRP *provides the best requested load.* The arithmetic mean improvement of FCFS × HRP is 1.003 and of FCFS × HRR is 1.000. Over the 5,400 sets of user and task behaviors there is little difference with respect to requested load for these three batchactive schedulers. Areas of difference are examined by varying individual parameters below.

Note that Figures 6.32 and 6.33 do not include simulations of large task sets which are covered in the per-parameter investigations below. Large task sets favor more intelligent disclosed queue subpolicies, and thus these improvement factors are conservative.

I now show the effects of varying individual parameters. I start with varying the number of users while holding other parameters constant. This parameter had the least effect but is shown for completeness. The following parameters, the number of tasks per task set and the task set change probability, better illuminate differences among the disclosed queue subpolicies.

I varied the number of users to show its effect on mean visible response time in Figure 6.34. FCFS × HRR does worse than the alternatives. It favors users who have requested more work, but in doing so, will eventually run their speculative tasks that will not be needed. (In effect, the spring from Figure 5.13 stretches too far.) FCFS × HRP does best by favoring users who have historically needed more of their speculative work.

Figures 6.35 and 6.36 show the effect of the number of users on requested load (charged load) and uncharged load (total load minus requested load), respectively. When the number of users is low, there is little or no competition for server resources. There is no difference among the schedulers

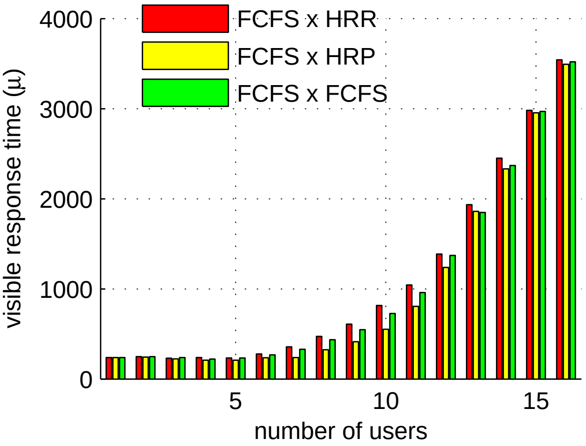


Figure 6.34: The effect of the number of users on batchactive usage of FCFS × HRR, batchactive usage of FCFS × HRP, and batchactive usage of FCFS × FCFS for mean visible response time. Each set of three scheduling configurations was run with a different number of users simultaneously competing for the shared resource while other parameters were held constant. Since lower visible response time is better, bars of less height indicate better performance. The differences are slight, but still, FCFS × HRR *provides worse mean visible response time while* FCFS × HRP *provides better mean visible response time.*

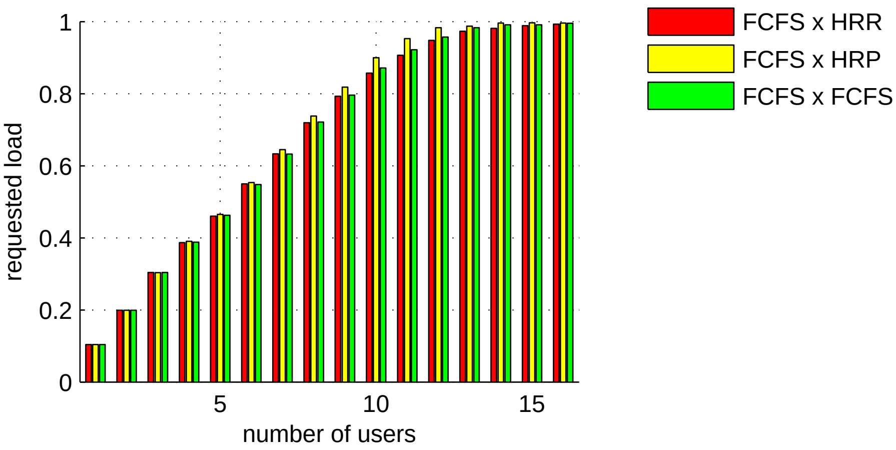


Figure 6.35: The effect of the number of users on batchactive usage of FCFS × HRR, batchactive usage of FCFS × HRP, and batchactive usage of FCFS × FCFS for requested load. Since higher requested load is better, bars of greater height indicate better performance. As the number of users increases, the HRP disclosed queue subpolicy begins providing slightly more requested load (server revenue). *Charged load between* FCFS × FCFS *and* FCFS × HRR *is nearly the same with* FCFS × HRP *outperforming both.*

because there are too few users available to choose from. The little induced uncharged load is from running slightly more work than these few numbers of users will eventually need. As the number of users increases, FCFS × HRP provides better requested load in the same region as it provided better mean visible response time (Figure 6.34). Favoring the right users enables the users to submit needed work more quickly, improving server revenue. It is in this region that FCFS × HRP achieves its greatest performance improvement for uncharged load. As the number of users increases further, there is always needed work to run and for all three configurations requested load converges to maximum and uncharged load converges to minimum.

Recall that it might not always be the case that it costs the resource provider the same for a server to be idle compared to a server to be busy (Chapter 5.1.3). The uncharged load, only present with batchactive scheduling due to the batchactive pricing mechanism, reflects the fraction of server busy time that goes uncharged. As shown in Figure 6.36, most of the time the uncharged load is small. In the worst case, it is roughly 20%. This frequency and degree of uncharged load was also found when varying other parameters (not shown) besides the number of users. Based on the increased cost

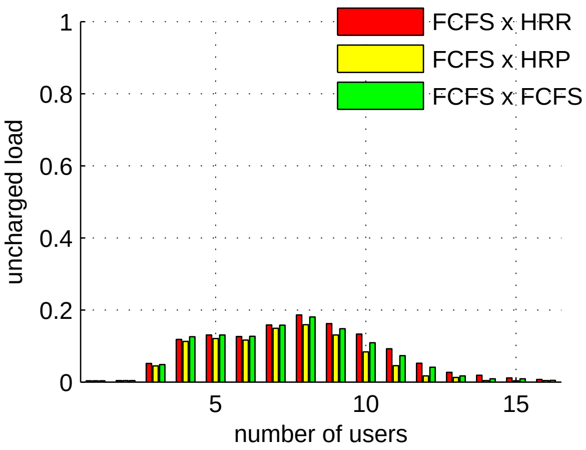


Figure 6.36: The effect of the number of users on batchactive usage of FCFS × HRR, batchactive usage of FCFS × HRP, and batchactive usage of FCFS × FCFS for uncharged load. Uncharged load is (total) load minus requested load. Since lower uncharged load is better, bars of less height indicate better performance. The middle range of users shows the greatest difference for this metric. Here, FCFS × HRP *performs best.* (The vertical axis ranges up to 1 since this is the maximum value uncharged load may take. This visually diminishes the maximum percentage difference between the schedulers which is approximately a factor of 2 at 11 users.)

to operate a busy as opposed to idle server, a resource provider may wish to increase the price for requested resources. (I do not advocate charging a reduced amount for unneeded speculative work because this would dissuade users from disclosing speculative tasks.) The effect of increased price on server revenue and user costs was discussed in Chapter 6.2.2.

I show the relation between requested load and mean visible response time by graphing both on the horizontal and vertical axes, respectively, in Figure 6.37. FCFS × HRP provides better values for these metrics in pairs than FCFS × FCFS. Further, FCFS × FCFS performs better than FCFS × HRR.

I show the relation between visible throughput and mean visible response time by graphing both on the horizontal and vertical axes, respectively, in Figure 6.38. Again, FCFS × HRP simultaneously provides the best visible throughput and mean visible response time.

Now I vary the upper bound of the number of tasks per task set, holding the other parameters constant.

I varied the number of tasks per task set to show its effect on mean visible response time in Figure 6.39. When all task sets consist of one task,

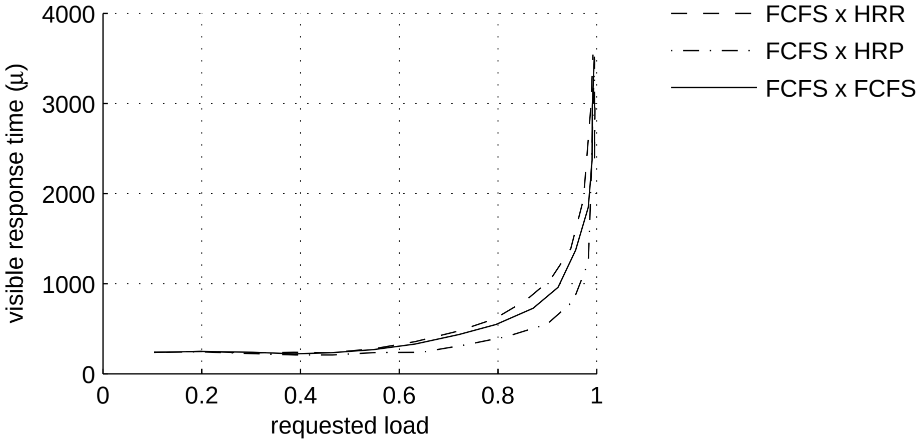


Figure 6.37: The relationship on batchactive usage of FCFS × HRR, batchactive usage of FCFS × HRP, and batchactive usage of FCFS × FCFS between requested load and mean visible response time as the number of users was varied from 1 to 16. Both axes are dependent axes; the horizontal axis indicates requested load while the vertical axis indicates mean visible response time. The number of users was varied while other parameters were held constant. The more a proposed configuration is below and to the right of FCFS × FCFS, the better its performance, as that indicates better requested load at better mean visible response time. FCFS × HRP *performs better for pairs of requested load and mean visible response time.* The setting of the independent variable, the number of users, may be found by examining Figures 6.34 and 6.35.

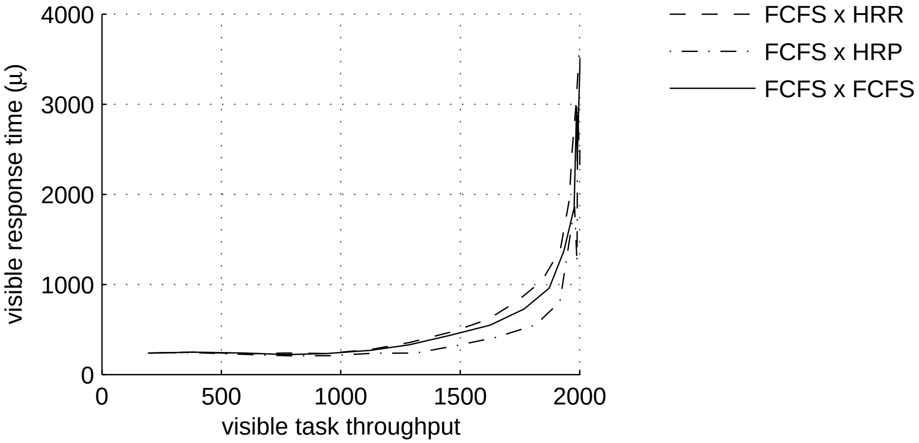


Figure 6.38: The relationship on batchactive usage of FCFS × HRR, batchactive usage of FCFS × HRP, and batchactive usage of FCFS × FCFS between visible throughput and visible response time as the number of users was varied from 1 to 16 and other parameters were held constant. Both axes are dependent axes; the horizontal axis indicates visible throughput while the vertical axis indicates mean visible response time. FCFS × HRP *provides better visible task throughput while simultaneously providing better mean visible response time.*

performance is identical across schedulers. As task set size increases, there is the opportunity to run disclosed work during user think time. The performance of all three schedulers improves. When the average task set size is too big, the returns diminish: the two-tiered batchactive schedulers' disclosed queues become swamped with unneeded work, lessening their benefits. However, FCFS × HRP is significantly less affected by task set sizes in the middle region of this graph, indicating that it is more robust.

I varied the number of tasks per task set to show its effect on requested load in Figure 6.40. Server revenue is surprisingly constant through a large range of task set sizes while a great difference in mean visible response time was presented in Figure 6.39. The region in which FCFS × HRP provided much better mean visible response time is the region in which it provides slightly better requested load, indicating that users are submitting needed work more quickly.

Now I vary the upper bound of the task set change probability, holding the other parameters constant.

I varied the task set change probability to show its effect on mean visible response time in Figure 6.41. The mean visible response of all three config-

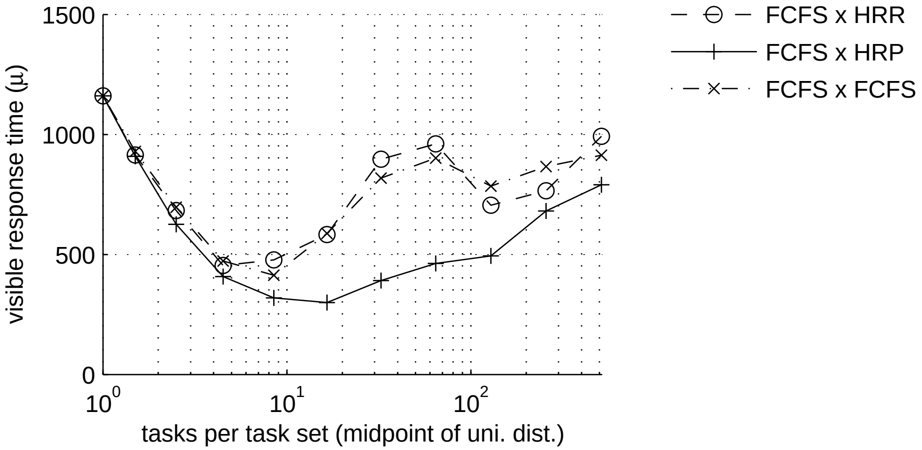


Figure 6.39: The effect of the number of tasks per task set on batchactive usage of FCFS × HRR, batchactive usage of FCFS × HRP, and batchactive usage of FCFS × FCFS for mean visible response time. Each set of three configurations was run with a different upper bound on the uniform distribution determining the number of tasks per task set for each user; the lower bound was held constant at 1. Other parameters were held constant. The number of tasks per task set is a uniform random variable and shown on the horizontal axis is the average of its lower and upper bounds. The saddle points at the horizontal axis value of 128 is within confidence intervals (not shown). FCFS × HRP *performs significantly better in the middle region of this graph, exhibiting greater robustness.* (This graph is log-linear.)

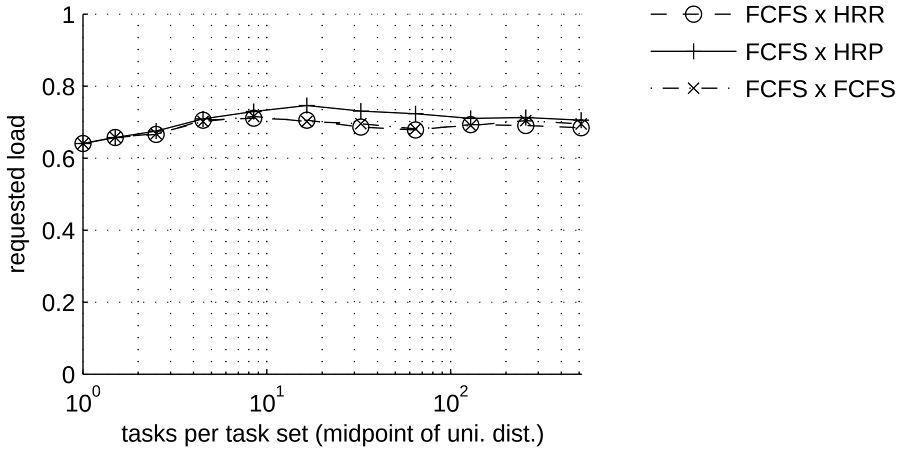


Figure 6.40: The effect of the number of tasks per task set on batchactive usage of FCFS × HRR, batchactive usage of FCFS × HRP, and batchactive usage of FCFS × FCFS for requested load. There is little difference in server revenue among the candidates for batchactive scheduling as task set sizes vary greatly. *The region in which* FCFS × HRP *performs slightly better is the region in which it provides the much better mean visible response time* in Figure 6.39.

urations worsens as users become more speculative. The relative difference between them (with FCFS × HRP better than FCFS × FCFS and FCFS × FCFS better than FCFS × HRR) remains unchanged.

The effect of the task set change probability on requested load and uncharged load also maintains the same ratio of performance differences between the candidate batchactive schedulers, thus the data is omitted. FCFS × HRP was better than FCFS × FCFS and FCFS × FCFS was better than FCFS × HRR.

This section showed the benefits of FCFS × HRP. FCFS × HRP learns user behavior; specifically, the likelihood of task request. The user model does not change how speculative users are, while in reality, users may be more or less certain of needing tasks at different times. More elaborate filters to learn behavior (such as the flip-flop filter discussed in Chapter 4.7) may perform well for such users. In any case, when predictions cannot be made, the batchactive system may fall back on FCFS × FCFS which still provides substantial benefits over non-speculative scheduling.

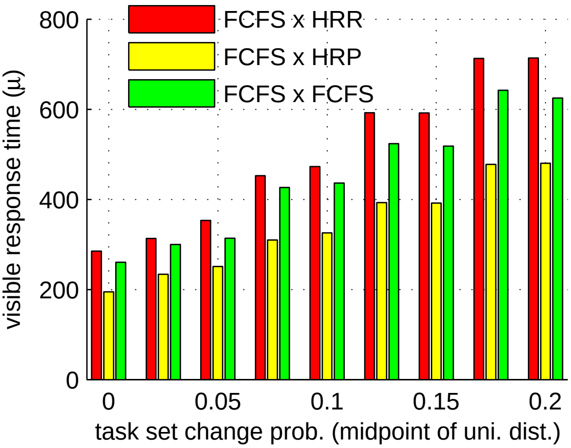


Figure 6.41: The effect of the task set change probability on batchactive usage of FCFS × HRR, batchactive usage of FCFS × HRP, and batchactive usage of FCFS × FCFS for mean visible response time. Each set of three configurations was run with a different upper bound on the uniform probability that determined the likelihood that users would cancel unviewed task outputs after thinking about received outputs. Other parameters were held constant. The probability is a uniform random variable and shown on the horizontal axis is the average of its lower (always 0) and upper bounds. Again, FCFS × HRP is the best scheduler. *Favoring the user who has requested the most disclosed tasks does best over different task set change probabilities.*

6.2.6 Benefits of favoring the speculative tasks of better speculators

This section compares non-speculative FCFS against batchactive FCFS × HRP, which was shown to perform better than FCFS × FCFS in the previous section (Chapter 6.2.5). I demonstrate the improvement of size-agnostic two-tiered batchactive scheduling using the novel disclosed queue subpolicy HRP over non-speculative FCFS.

Data on mean scaled billed resources, which is only applicable to batch usage of a non-speculative scheduler, can be found in Chapter 6.2.4.

Concerning time-based metrics, the improvement factors of mean visible response time and mean visible slowdown are shown in Figures 6.42 and 6.43. The general shape of these curves are very similar to Figures 6.6 and 6.7, but there is slightly more area under these curves, indicating that the improvement of FCFS × HRP over batch and interactive usage of FCFS is better than the improvement of FCFS × FCFS. More detail is found in the per-parameter investigations below.

Concerning the cost-based metric, the improvement factors of requested load are shown in Figure 6.44. Requested load is usually better for batch users on a non-speculative scheduler. That is, a resource provider would earn more when all users speculate by requesting task sets and all users are charged for any resource usage. This is an unlikely scenario, however; it assumes that all users are highly confident of needing all speculative tasks immediately or that all users have abundant resources to pay for unneeded speculative tasks. Further, these users would experience significantly worse mean visible response time at the higher levels of server revenue. More detail is found in the per-parameter investigations below.

Again, because Figures 6.42, 6.43, and 6.44 do not include simulations of large task sets, which favor batchactive scheduling, and which are covered below, these summarizing graphs are conservative.

For the runs covered in the above experiments, Table 6.7 reports the number of deadlines met. Interactive FCFS cannot meet deadlines because tasks are requested only after needed. Batch FCFS surprisingly meets more deadlines than FCFS × HRP. This is because FCFS × HRP shifts attention to requested tasks in an attempt to minimize visible response time while batch FCFS continually aids tasks until they complete.

Table 6.8 confirms that the variance of visible response time for the runs covered in the experiments above does not become worse for FCFS × HRP batchactive scheduling. In fact, it improves under batchactive scheduling as tasks that are not known to be needed are deferred when known-needed work is present and lower visible response times are achieved.

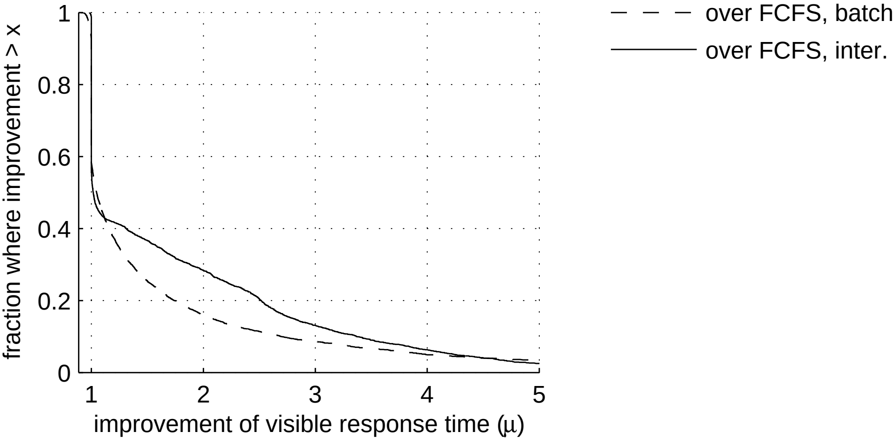


Figure 6.42: Improvement of batchactive usage of FCFS × HRP over interactive and batch usage of FCFS for mean visible response time. A task's visible response time is the time between a user needing and receiving the task's output. The vertical axis indicates the fraction of the runs exploring 5,400 sets of parameters in which the improvement was *at least* as much indicated on the horizontal axis. Batchactive performance is measured by the area under the curves for horizontal axis values greater than 1 minus the area above the curves for horizontal axis values between 0 and 1. Thus, FCFS × HRP *performs at least twice as well for about* 17% *and* 30% *of the simulated behaviors of batch* FCFS *and interactive* FCFS, respectively. The arithmetic mean improvement is 1.731 over interactive FCFS and 1.583 over batch FCFS.

FCFS, batch	FCFS, interactive	FCFS × HRP
791.610 (99.690)	0.000 (0.000)	206.553 (20.590)

Table 6.7: The number of deadlines met among batch usage of FCFS, interactive usage of FCFS, and batchactive usage of FCFS × HRP. A deadline for a task is met if it has executed before the user needs its output. Therefore, a higher deadlines met is better. The mean of all runs for each scheduler is presented. The 95% confidence interval of each mean is the mean plus and minus the value in parenthesis. FCFS × HRP *performs better than interactive* FCFS *and worse than batch* FCFS. Interactive FCFS never runs a task before needed (its 'deadline'). Batch FCFS continually aids tasks until they complete. FCFS × HRP performs better than FCFS × FCFS from Table 6.4.

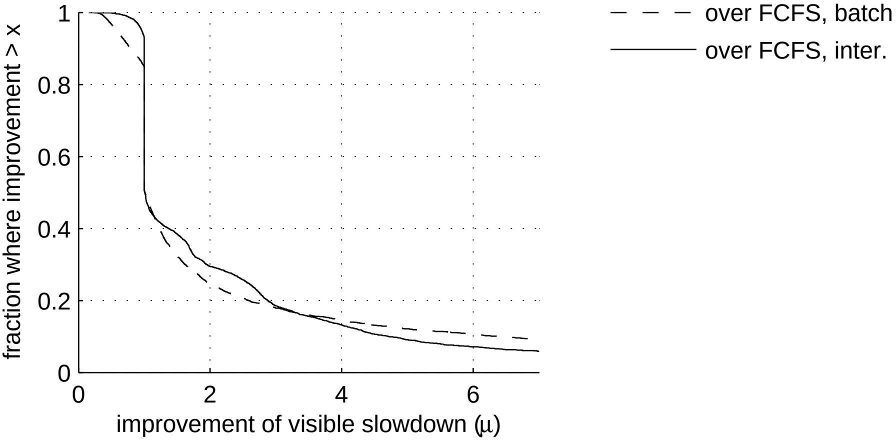


Figure 6.43: Improvement of batchactive usage of FCFS × HRP over interactive and batch usage of FCFS for mean visible slowdown. A task's visible slowdown is its visible response time scaled by task size. Thus, FCFS × HRP *performs at least twice as well for about* 25% *and* 30% *of the simulated behaviors of batch* FCFS *and interactive* FCFS, respectively. The arithmetic mean improvement is 2.547 over interactive FCFS and 3.625 over batch FCFS. The mean improvement is higher against batch FCFS because of its heavier tail.

FCFS, batch	FCFS, interactive	FCFS × HRP
20,182.380 (755.277)	4,542.007 (101.280)	4,382.339 (103.666)

Table 6.8: The standard deviation of visible response time among batch usage of FCFS, interactive usage of FCFS, and batchactive usage of FCFS × HRP. This metric reflects how different visible response times are. Users dislike variability; lower standard deviations are better. The mean of all runs for each scheduler is presented. The 95% confidence interval of each mean is the mean plus and minus the value in parenthesis. *Batchactive scheduling improves the variance of visible response time.*

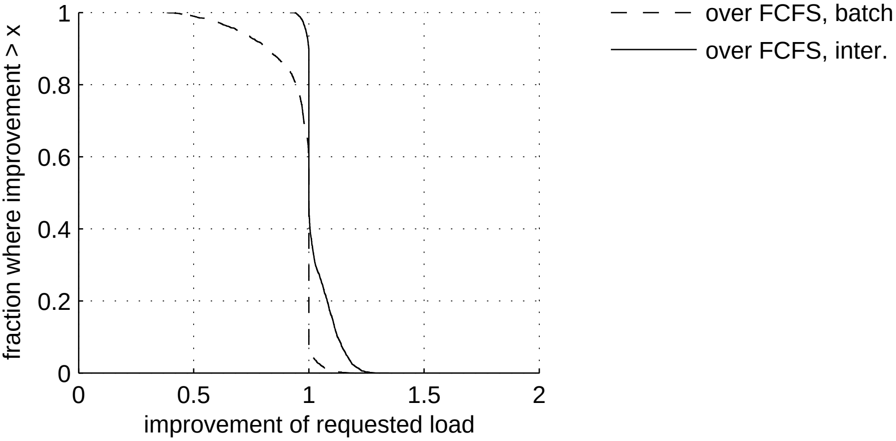


Figure 6.44: Improvement of batchactive usage of FCFS × HRP over interactive and batch usage of FCFS for requested load. Requested load is load for the non-speculative configurations. For FCFS × HRP, requested load is the load actually charged. The area under the curves for improvement greater than 1 reflects better batchactive performance, while the area above the curves for improvement less than 1 reflects worse performance. FCFS × HRP *provides better requested load than interactive* FCFS *and worse requested load than batch* FCFS. As was explained in Chapter 6.2.2, it is expected that batchactive scheduling does worse than batch usage of FCFS for requested load. The arithmetic mean improvement is 1.033 over interactive FCFS and 0.955 over batch FCFS. (A mean improvement factor less than 1 is overall worse performance.)

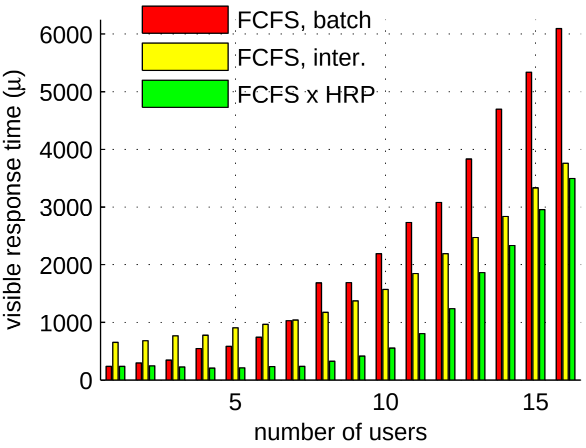


Figure 6.45: The effect of the number of users on batchactive usage of FCFS × HRP, interactive usage of FCFS, and batch usage of FCFS for mean visible response time. Each set of three scheduling configurations was run with a different number of users simultaneously competing for the shared resource while other user and task characteristics were held constant. The different configurations are represented by bars of different shades, with the right-most bar being the batchactive configuration. Since lower visible response time is better, bars of less height indicate better performance. FCFS × HRP *adapts over the range of users and always performs best.*

I now show the effects of varying individual parameters. I start with varying the number of users while holding other parameters constant.

I varied the number of users to show its effect on mean visible response time in Figure 6.45. The description of FCFS × FCFS performance in Figure 6.10 applies here as well. The only difference is that batchactive improvements are slighter better because FCFS × HRP is better than FCFS × FCFS.

I show that a higher number of needed tasks get through the system at an acceptable visible response time by graphing visible throughput on the horizontal axis and mean visible response time on the vertical axis in Figure 6.46. The greater distance below and to the right that the batchactive curve for FCFS × HRP has in this graph compared to the distance that FCFS × FCFS had in Figure 6.15 shows the advantage of HRP as the disclosed queue subpolicy.

The demonstration of more billable time at given mean visible response times by graphing requested load on the horizontal and mean visible response time on the vertical axis is similar to Figure 6.16 and is thus omitted.

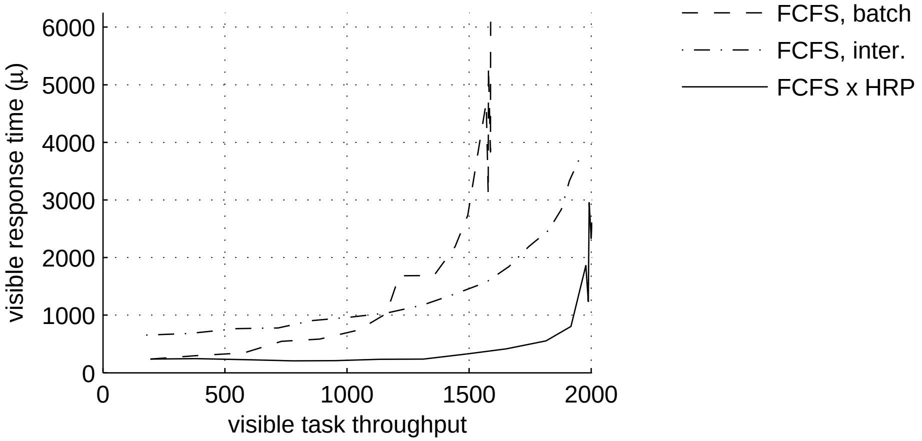


Figure 6.46: The relationship on batchactive usage of FCFS × HRP, interactive usage of FCFS, and interactive usage of FCFS between visible throughput and visible response time as the number of users was varied from 1 to 16. Both axes are dependent axes; the horizontal axis indicates visible task throughput while the vertical axis indicates mean visible response time. FCFS × HRP *always provides better pairs of visible throughput and mean visible response time.* At a mean visible response time of $1,000$, the needed tasks that finished for FCFS × HRP was about 60% higher against both alternatives. The number of users was varied while other parameters were held constant. The more the batchactive curve is below and to the right of the other curves, the better its performance, as that indicates better visible throughput at better mean visible response time.

The improvements and description of how the number of users affects load, requested load, and visible throughput are also similar to those for Figures 6.11, 6.12, and 6.14, and thus data and discussion are omitted.

To follow the form of the previous sections, I would now vary the upper bound of the task set change probability, holding the other parameters constant. However, again, the improvements and description of how this parameter affects metrics such as mean visible response time and requested load mirror those in Figures 6.17 and 6.19, and thus data and discussion are omitted.

I now vary the upper bound of the number of tasks per task set, holding the other parameters constant.

I varied the number of tasks per task set to show its effect on visible response time in Figure 6.47. When all task sets have only one task, all cases provide the same mean visible response time. Interactive FCFS is immune to the number of tasks per task set because these users will have submitted at most one task from their task sets. Batch FCFS and FCFS × HRP initially improve with speculative work that may be performed while users are in their think times. Soon batch FCFS becomes unusable as its single queue becomes overwhelmed. FCFS × HRP is resilient to large task sets. Not only does it perform significantly better than batch FCFS, but it also performs better than batchactive FCFS × FCFS from Figure 6.22. Eventually its benefits decline as the disclosed queue becomes swamped with unneeded tasks.

FCFS × HRP also provides advantages over FCFS × FCFS for requested load, but the differences are minor, and thus I do not present data showing how task set size affects requested load.

To follow the form of the previous sections, I would vary the mean service time, holding other parameters constant and then vary the mean think time, holding other parameters constant. However, the improvements and description of how these parameters affect metrics such as mean visible response time and requested load mirror those in Figures 6.25, 6.26, 6.27, and 6.29, and thus data and discussion are omitted.

This section repeated some of the experiments from the first section that demonstrated batchactive superiority (Chapter 6.2.4) with the better disclosed queue subpolicy, HRP, studied in the previous section (Chapter 6.2.5). FCFS × HRP was shown to have better resiliency for large task sets, provide better mean visible response time and slightly better requested load, among other metrics. Further, no performance downside to FCFS × HRP (compared to FCFS × FCFS's performance against non-speculative scheduling) was found.

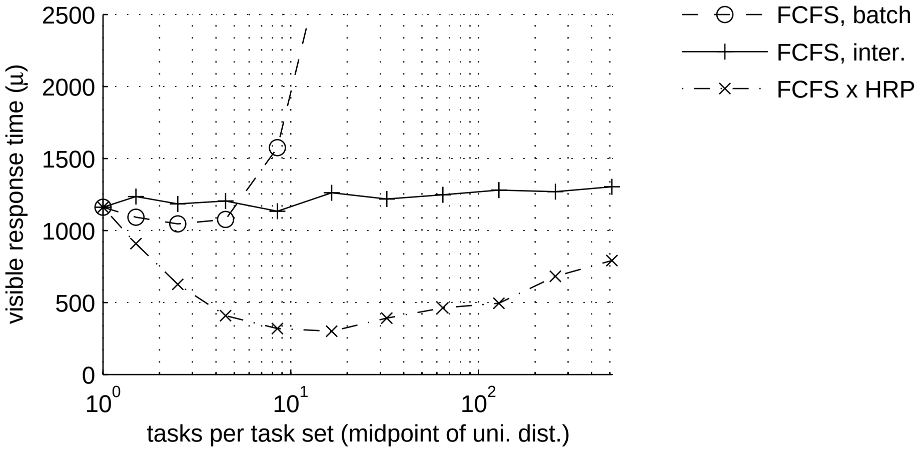


Figure 6.47: The effect of the number of tasks per task set on batchactive usage of FCFS × HRP, interactive usage of FCFS, and batch usage of FCFS for mean visible response time. Each set of three configurations was run with a different upper bound on the uniform distribution determining the number of tasks per task set for each user; the lower bound was always held at 1. Other parameters were held constant. The number of tasks per task set is a uniform random variable and shown on the horizontal axis is the average of its lower and upper bounds. *The better performance of* FCFS × HRP *in this graph compared to* FCFS × FCFS *from Figure 6.22 demonstrates the benefit of* HRP *as the disclosed queue subpolicy for large task sets.* (This graph is log-linear.)

6.2.7 Benefits of two-tiered usage-based scheduling

In this section I compare non-speculative user-FB against batchactive user-requested-FB × HRP. Interactive user-FB is not explored because usage-based scheduling is usually employed when users are not directly charged for resource usage (such as in communal cost-centers), and thus users have no motivation to throttle their speculation. I suspect that the relative performance improvement of batchactive user-requested-FB × HRP over interactive user-FB would be similar to the improvement of batchactive FCFS × HRP over interactive FCFS × FCFS (Figure 6.42). Moreover, mean scaled billed resources and requested load are omitted, again because usage-based scheduling is usually employed when resources are not directly charged.

The improvement factors of mean visible response time and mean visible slowdown are shown in Figures 6.48 and 6.49, respectively. Batchactive improvements on mean visible response time are not as pronounced when the requested queue scheduling policy is usage-based compared to using FCFS for the requested queue in Figure 6.6. However, the improvements on mean visible slowdown are still significant. Slowdown reflects the performance of the majority of smaller tasks better than response time. More detail is found in the per-parameter results below.

Again, because Figures 6.48 and 6.49 do not include simulations of large task sets, which favor batchactive scheduling, these summarizing graphs are conservative.

Some users, such as those with bigger task sets, will use more resources than others when there is low contention (since all the schedulers under consideration are work-conserving). Still, it is important to show that batchactive scheduling will not worsen the variance of user requested resource usage when usage-based scheduling is employed compared to non-speculative scheduling. If it did, then it would not correctly police resource usage when resources are not directly charged, which is one motivation for usage-based scheduling. Table 6.9 confirms this; in fact, users use a more equal amount of requested resources under batchactive scheduling.

I now show the effects of varying the number of users while holding other parameters constant. Varying other parameters (not shown) did not provide insights.

I varied the number of users to show its effect on mean visible response time in Figure 6.50. Interestingly, while batchactive performance for user-requested-FB × HRP is similar to FCFS × FCFS from Figure 6.10, batch user-FB is much better than batch FCFS from that figure. This is because favoring users who have requested less helps defer the long-shot speculation

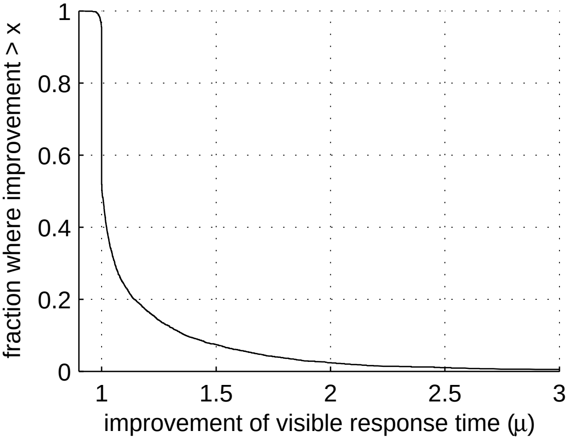


Figure 6.48: Improvement of batchactive usage of user-requested-FB × HRP over batch usage of user-FB for mean visible response time. A task's visible response time is the time between a user needing and receiving the task's output. The horizontal axis shows improvement factors while the vertical axis shows the fraction of the runs in which the improvement was *at least* as much indicated on the horizontal axis. Batchactive performance is measured by the area under the curve for improvement values greater than 1 minus the area above the curve for improvement values between 0 and 1. *For* 10% *of the runs, batchactive user-requested-*FB × HRP *performs at least* 1.5 *times better.* The improvements are not as dramatic as when FCFS is used for the requested queue (Figure 6.6). The arithmetic mean improvement is 1.121.

user-FB, batch	user-requested-FB × HRP
6,814.556 (383.948)	4,203.379 (133.723)

Table 6.9: The standard deviation of user requested resource usage among batch usage of user-FB and batchactive usage of user-requested-FB × HRP. This metric reflects how close users come to using an equal amount of resources, a goal when resource usage is not directly charged to prevent resource abuse. This table shows that batchactive scheduling does not make the variance of user requested resource usage worse while improving metrics such as mean visible response time. In fact, *batchactive scheduling improves the variance of user requested resource usage.* The mean of all runs for each scheduler is presented. The 95% confidence interval of each mean is the mean plus and minus the value in parenthesis.

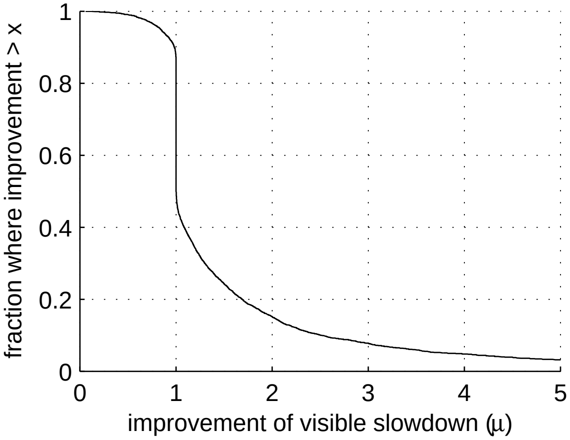


Figure 6.49: Improvement of batchactive usage of user-requested-FB × HRP over batch usage of user-FB for mean visible slowdown. A task's visible slowdown is its visible response time scaled by task size. Many hold that slowdown is more important than response time because it better reflects the performance of the majority of small tasks, and because it reflects the expectation that larger tasks should take longer to run (Chapter 4.4.1). *For* 20% *of the runs, batchactive user-requested-*FB × HRP *performs at least* 1.5 *times better.* The arithmetic mean improvement is 1.715.

deep in someone's task set in favor of the needed tasks at the top of the task sets of other users. Still, although batch usage benefits from a usage-based non-speculative scheduling, it is still worse than batchactive usage of a batchactive scheduler.

This section showed the applicability of batchactive scheduling to environments in which usage-based scheduling is employed. Performance improvements still occur when the requested queue is serviced by user-requested-FB.

6.2.8 Benefits of two-tiered SRPT

Here I examine the performance of size-aware policies based on SRPT. Techniques for predicting task size when unknown were discussed in Chapter 4.7. Specifically, I compare non-speculative SRPT v. batchactive SRPT × FCFS. These results are similar to those in Chapter 6.2.4 comparing FCFS × FCFS against FCFS, which does not rely on task size knowledge.

Concerning time-based metrics, the improvement factors of mean visible response time and mean visible slowdown are shown in Figures 6.51 and 6.52, respectively. Batchactive scheduling provides better mean visible

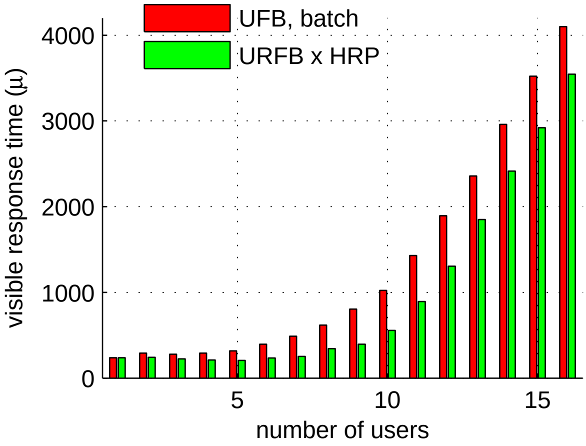


Figure 6.50: The effect of the number of users on batchactive usage of user-requested-FB × HRP and batch usage of user-FB for mean visible response time. Mean visible response time is the average time users are blocked waiting for needed task output. The number of users simultaneously competing for the shared resource was varied while other parameters were held constant. The policy *user-requested*-FB × HRP *improves with more users.* Batch usage of user-FB is always worse but better than the batch case when using FCFS in Figure 6.10.

response time and mean visible slowdown on average than non-speculative scheduling. The mean visible response time improvements are similar to those shown with non-size-based scheduling in Figure 6.6, while the improvements for mean visible slowdown are not as pronounced as compared to those in Figure 6.7. The reasons for batchactive improvements are examined in the per-parameter investigations below.

Concerning cost-based metrics, the charges for unneeded speculation incurred by batch users of the non-speculative scheduler are shown in Figure 6.53 and the improvement factors of requested load (reflecting server revenue) are shown in Figure 6.54. Batchactive users always pay less than uses submitting batches of work to non-speculative schedulers. Interestingly, batch users pay even more for unneeded speculation here than under the non-size-based scheduler in Figure 6.8: the better the scheduling, the more needless work completed. A batchactive system often generates slightly more revenue than a non-speculative system populated by users who never submit unneeded work (viz., the interactive users). A batchactive system usually generates less revenue than a non-speculative system populated by those users who pay more by submitting batches of work, some of which will not

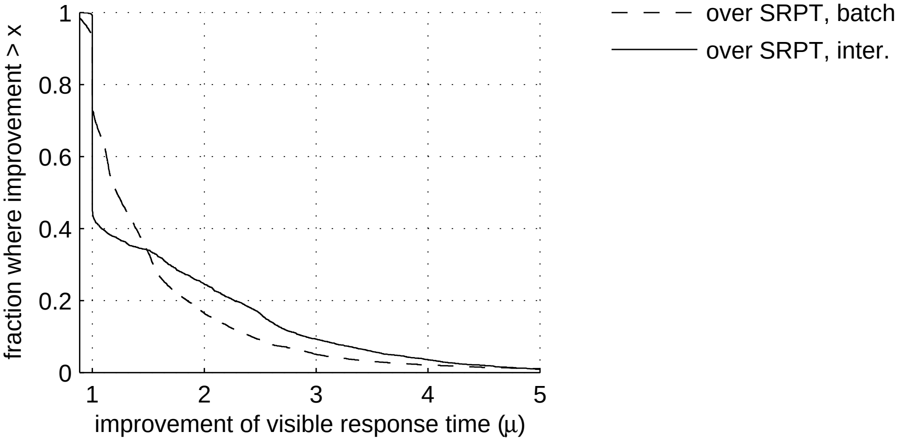


Figure 6.51: Improvement of batchactive usage of SRPT × FCFS over interactive and batch usage of SRPT for mean visible response time. A task's visible response time is the time between a user needing and receiving the task's output. The horizontal axis shows improvement factors, while the vertical axis indicates the fraction of the runs in which the improvement was *at least* as much indicated on the horizontal axis. Cases in which improvements occurred are shown by the area under the curves for horizontal axis values greater than 1, whereas cases in which batchactive scheduling did worse are shown by the area above the curves for horizontal axis values between 0 and 1. *Performance was at least* 1.5 *times better than both batch and interactive* SRPT *for* 35% *of the runs.* The arithmetic mean improvement is 1.579 over interactive SRPT and 1.522 over batch SRPT.

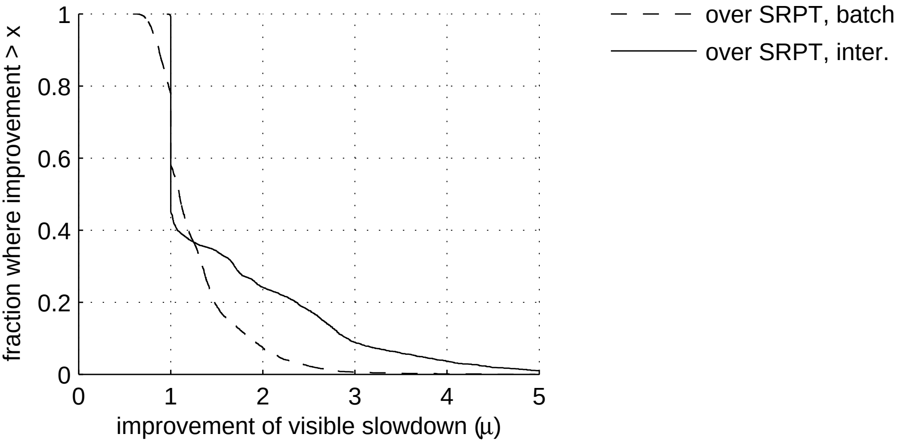


Figure 6.52: Improvement of batchactive usage of SRPT × FCFS over interactive and batch usage of SRPT for mean visible slowdown. A task's visible slowdown is its visible response time divided by its size, reflecting the expectation that bigger tasks should take longer to run. The horizontal axis shows improvement factors, while the vertical axis shows the fraction of runs in which the improvement was *at least* as much indicated on the horizontal axis. *Performance was at least* 1.5 *times better than batch* SRPT *for* 20% *of the runs and also at least* 1.5 *times better than interactive* SRPT *for* 35% *of the runs.* The arithmetic mean improvement is 1.590 over interactive SRPT and 1.252 over batch SRPT.

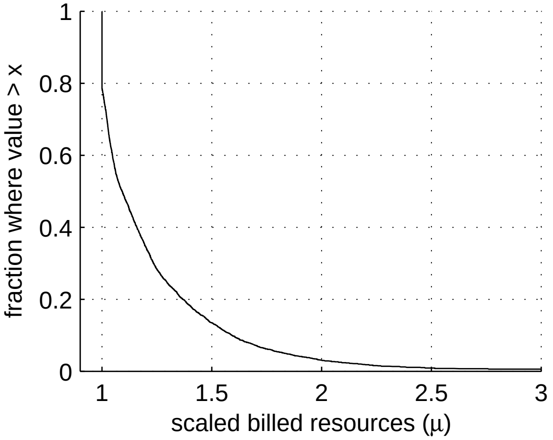


Figure 6.53: Mean scaled billed resources for batch usage of SRPT. For one user, scaled billed resources is the amount of resources charged over resources needed. Batch users often pay for more resources than they need. The horizontal axis shows mean scaled billed resources while the vertical axis indicates the fraction of runs in which the mean scaled billed resources was at least as much indicated on the horizontal axis. More area under the curve indicates higher costs for batch users. *The average batch user using* SRPT *pays at least* 40% *more than necessary for* 20% *of the runs.* Over all runs, the average mean scaled billed resources is 1.261.

be needed (viz., the batch users). These improvements are very similar to those for the non-size-based scheduling in Figure 6.9. Again, it is unlikely that users will behave in a batch manner: it assumes that all users are highly confident of needing all speculative tasks immediately or that all users have the resources to pay for unneeded speculation. Further, these users would experience significantly worse mean visible response time at the higher levels of server revenue. Below I vary individual parameters to show the conditions under cost-based metrics differ most.

Again, because Figures 6.51, 6.52, 6.53, and 6.54 do not include simulations of large task sets, which favor batchactive scheduling, and which are covered below, these summarizing graphs are conservative.

I now show the effects of varying individual parameters. I kept the vertical axis ranges the same with some of the graphs in Chapter 6.2.4 to illustrate that SRPT-based schedulers provide better absolute performance than FCFS-based schedulers. Still the relative difference between batchactive and non-speculative schedulers based on FCFS or SRPT is similar, as shown by the improvement graphs above.

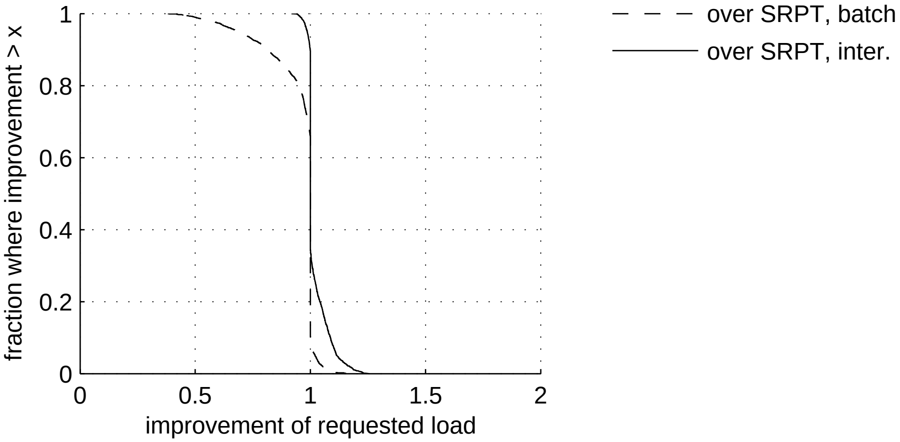


Figure 6.54: Improvement of batchactive usage of SRPT × FCFS over interactive and batch usage of SRPT for requested load. Requested load is the server busy time charged to the user. The horizontal axis shows improvement factors while the vertical axis indicates the fraction of the runs in which the improvement was *at least* as much indicated on the horizontal axis. Better batchactive performance is reflected by the area under the curves for improvement values greater than 1, while worse batchactive performance is reflected by the area above the curves for improvement values less than 1. Data along the line where improvement equals 1 reflects equal performance for some user and task characteristics. SRPT × FCFS *provides better requested load than interactive* SRPT *and worse requested load than batch* SRPT*.* Doing worse against batch FCFS is expected and was explained in Chapter 6.2.2. The arithmetic mean improvement is 1.020 over interactive SRPT and 0.956 over batch SRPT. (A mean improvement factor less than 1 is overall worse performance.)

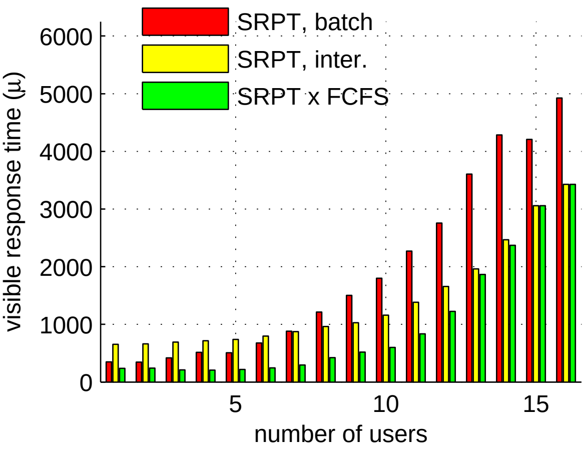


Figure 6.55: The effect of the number of users on batchactive usage of SRPT × FCFS, interactive usage of SRPT, and batch usage of SRPT for mean visible response time. Each set of three scheduling configurations was run with a different number of users simultaneously competing for the shared resource while other user and task characteristics were held constant. Visible response time for a task is the time a user was blocked on its output. The different configurations are represented by bars of different shades, with the right-most bar being the batchactive configuration. Since lower visible response time is better, bars of less height indicate better performance. *Batchactive scheduling adapts to different numbers of users, always providing the best mean visible response time.*

I start with varying the number of users while holding other parameters constant.

I varied the number of users to show its effect on mean visible response time in Figure 6.55. Nearly always, batchactive SRPT × FCFS performs best, exhibiting adaptability across the range of the numbers of users, while batch and interactive usage of SRPT trade off where they are best suited. The description follows that for non-size-based scheduling for Figure 6.10. The only difference is that overall mean visible response time is less for all configurations, as expected when using SRPT as the underlying scheduler (Chapter 4.5.1). Comparing Figures 6.10 and 6.55 shows that FCFS × FCFS even beats the non-speculative versions of SRPT, thus the availability of a task size oracle will not diminish the value of batchactive scheduling.

I varied the number of users to show its effect on mean visible slowdown in Figure 6.56. Batch SRPT is suited to few users as it pipelines speculative tasks with think time. Batchactive scheduling always achieves the best mean visible slowdown. With many users, the visible slowdowns of batch SRPT and

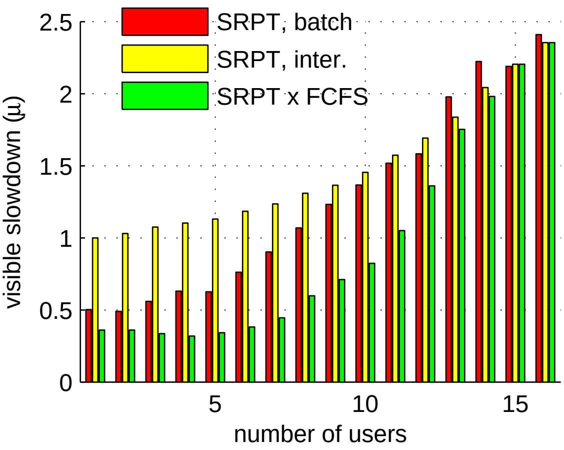


Figure 6.56: The effect of the number of users on batchactive usage of SRPT × FCFS, interactive usage of SRPT, and batch usage of SRPT for mean visible slowdown. Visible slowdown for a task is its visible response time scaled by its service time. Since lower visible slowdown is better, bars of less height indicate better performance. *Batchactive scheduling never performs worse than the alternatives for mean visible slowdown; its greatest improvement occurs with low to medium numbers of users.* (Recall from Chapter 5.2 that visible slowdown can be less than one.)

interactive SRPT converge, in contrast to the mean visible response time graph in Figure 6.55. This occurs because the visible response times of the large tasks affect visible slowdown less.

Mean scaled billed resources is not affected by varying the number of users, so this data is not presented.

The induced (total) load, requested load, and visible throughput when using size-based scheduling compared to FCFS-based scheduling (Figures 6.11, 6.12, and 6.14) is nearly the same, thus this data is omitted. The description presented earlier (Chapter 6.2.4) holds.

Almost always, batchactive scheduling provides better mean visible response time and better visible throughput. I now show the relation between these metrics for different runs for the three scheduling configurations. I show that a higher number of needed tasks get through the system at an acceptable visible response time by graphing visible throughput on the horizontal axis and mean visible response time on the vertical axis in Figure 6.57. While batchactive scheduling is able to simultaneously provide better values for these two metrics, the differences using SRPT-based scheduling is not as pronounced as using FCFS-based scheduling (Figure 6.15). This same phe-

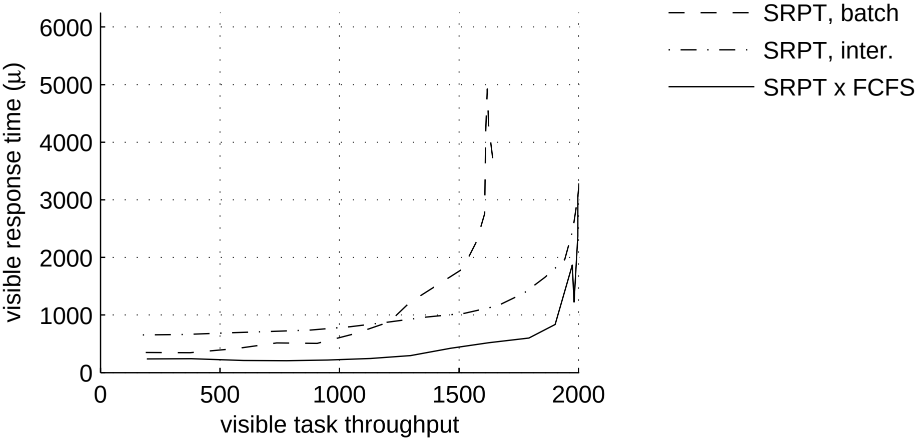


Figure 6.57: The relationship on batchactive usage of SRPT × FCFS, interactive usage of SRPT, and batch usage of SRPT between visible throughput and visible response time as the number of users was varied from 1 to 16. Both axes are dependent axes; the horizontal axis indicates visible task throughput while the vertical axis indicates mean visible response time. The number of users was varied while other parameters were held constant. The more the batchactive curve is below and to the right of the other curves, the better its performance, as that indicates better visible throughput at better mean visible response time. SRPT × FCFS *always provides better pairs of visible throughput and mean visible response time.* At a visible throughput of $1,500$, the visible response time of SRPT × FCFS is over 2 times better than interactive SRPT and over 4 times better than batch SRPT. The differences between batchactive and non-speculative scheduling in simultaneously providing better mean visible response time and mean visible throughput for SRPT-based scheduling is not as pronounced as for FCFS-based scheduling (Figure 6.15).

nomena holds for pairs of mean visible response time and requested load (not shown).

To follow the form of the previous sections, I would now vary the other parameters and present their affects on each metric. However, most results follow the form and description of non-size-based scheduling (Chapter 6.2.4), except that the absolute mean visible response times are lower using SRPT-based scheduling. The remaining graphs present interesting differences or reinforce central points.

I varied the number of tasks per task set to show its effect on mean visible slowdown in Figure 6.58. Mean visible slowdown begins to improve for batchactive SRPT × FCFS and batch SRPT as there becomes available speculative work with which to pipeline user think time. Quickly, the sin-

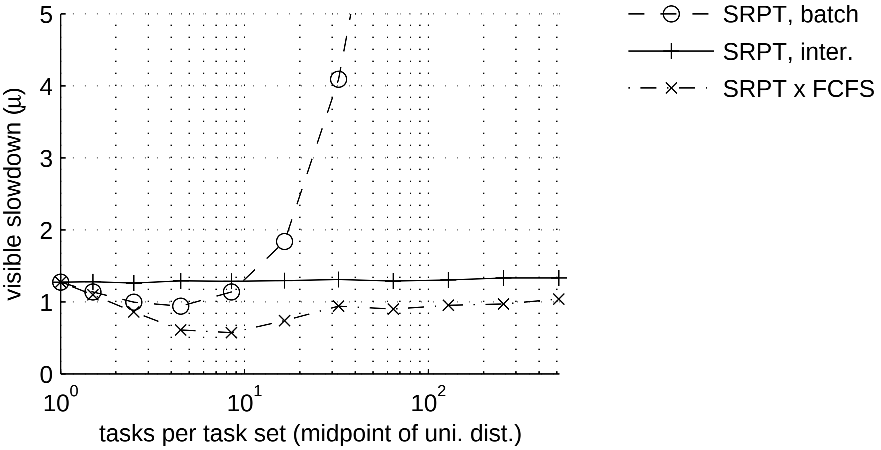


Figure 6.58: The effect of the number of tasks per task set on batchactive usage of SRPT × FCFS, interactive usage of SRPT, and batch usage of SRPT for mean visible slowdown. The number of tasks per task set is a uniform random variable and shown on the horizontal axis is the average of its lower and upper bounds. As the number of tasks per task set increases, both batch FCFS and FCFS × FCFS provide visible slowdowns less than one. Soon, batch FCFS's visible slowdown increases rapidly and is over 165 when the upper bound for the number of tasks per task set is $1,024$. SRPT × FCFS *always wins.* (This graph is log-linear.)

gle, requested queue of the non-speculative scheduler with batch users becomes overwhelmed and mean visible slowdown grows rapidly. Batchactive scheduling always provides the best mean visible slowdown. Eventually, it would converge to the performance of interactive SRPT as its disclosed queue becomes swamped with unneeded speculation.

I varied the number of tasks per task set to show its effect on mean scaled billed resources in Figure 6.59. Interestingly, by comparing the ranges of the vertical axes, one sees that the better scheduling of SRPT causes users to waste more money on unneeded speculation in contrast to the batch FCFS users in Figure 6.23.

To show that batchactive scheduling improves performance under distributions other than exponential distributions, I experimented with the Pareto distribution. Service time was taken from a Bounded Pareto with minimum observation 5 s, maximum observation $1,000,000$ s, and an α value of 1.0. Think time was taken from a Bounded Pareto with minimum observation 60 s, maximum observation $1,000,000$ s, and an α value of 1.0. These selections result in a theoretic mean service time of approximately 61 s and mean

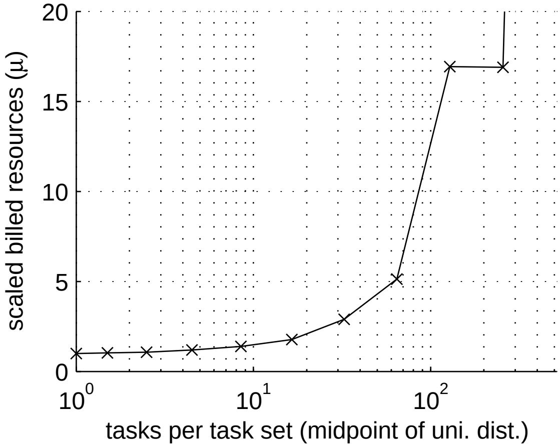


Figure 6.59: The effect of the number of tasks per task set on batch usage of SRPT for mean scaled billed resources. *Batch users spend more with the better scheduling provided by* SRPT *than* FCFS (Figure 6.23). (This graph is log-linear.)

think time of approximately 583 s, within the range of expected usage.

Using these Bounded Pareto distributions, the improvement factors of mean visible response time are shown in Figure 6.60 and the charges for unneeded speculation incurred by batch users of the non-speculative scheduler are shown in Figure 6.61. The differences between batchactive and non-speculative scheduling performance is not as pronounced; a parameter-by-parameter investigation of performance differences (not shown), especially for large task sets, would illuminate where the greatest benefits of batchactive scheduling are for Pareto-distributed service and think times.

This section showed that batchactive scheduling provides similar improvements when SRPT size-based scheduling is employed. The benefits of SRPT depend on the quality of task size prediction. I believe that inaccurate predictions would affect batchactive and non-speculative scheduling to the same extent. Gibbons [1997] demonstrates in simulation an up to 75% benefit of using predicted values of task service time in a scheduling policy that makes use of task service time compared to using actual service time. If predictions are not trusted, an alternative would be FB-based scheduling, which favors the task that has run the least amount of time. FB-based schedulers, which do not need task size, would give time-based performance between that of FCFS- and SRPT-based schedulers (Chapter 4.5.1).

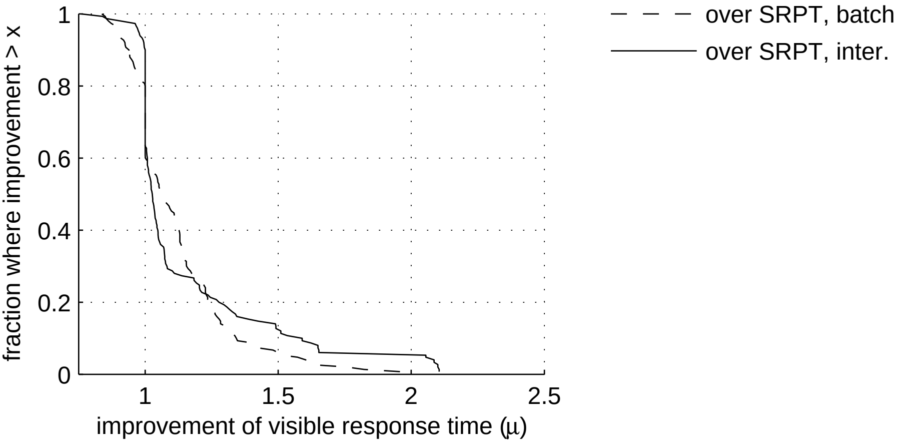


Figure 6.60: Improvement of batchactive usage of SRPT × FCFS over interactive and batch usage of SRPT for mean visible response time using Bounded Pareto distributions for task service time and user think time. The horizontal axis shows improvement factors, while the vertical axis indicates the fraction of the runs in which the improvement was *at least* as much indicated on the horizontal axis. *Performance was at least* 25% *better than both batch and interactive* SRPT *for* 20% *of the runs.* The arithmetic mean improvement of SRPT × FCFS is 1.151 over interactive SRPT and 1.116 over batch SRPT.

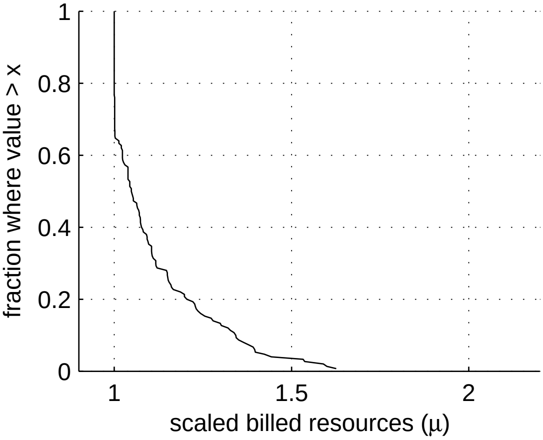


Figure 6.61: Mean scaled billed resources for batch usage of SRPT using Bounded Pareto distributions for task size and think time. For one user, scaled billed resources is the amount of resources charged over resources needed. Users who behave in a batch manner often pay for more resources than they need, as speculative tasks are billed and the user later determines their outputs are unneeded. SRPT × FCFS charges less (not shown) because disclosed tasks are not charged according to the batchactive pricing mechanism. *The average batch user using* SRPT *pays at least* 25% *more than necessary for* 18% *of the runs.* Over all runs, the average mean scaled billed resources is 1.107.

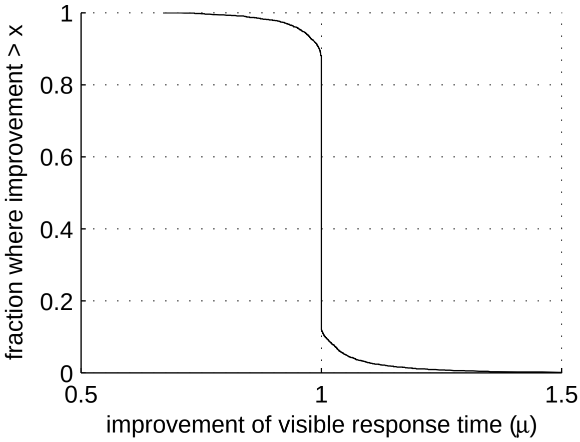


Figure 6.62: Improvement of SRPT × RFCFS over SRPT × HRP for mean visible response time. A task's visible response time is the time between a user needing and receiving the task's output. The horizontal axis shows improvement factors while the vertical axis indicates the fraction of the runs in which the improvement was *at least* as much indicated on the horizontal axis. *The similar areas below the curve for improvements greater than* 1 *and above the curve for improvements less than* 1 *suggest no significant difference.* In fact, the arithmetic mean improvement is 1.002.

6.2.9 Performance of an impractical disclosed queue subpolicy

This section shows that HRP performs well in the space of disclosed queue subpolicies for two-tiered batchactive scheduling; the relative performance of an impractical disclosed queue subpolicy, RFCFS, is nearly the same. RFCFS knows perfectly whether a speculative task will ever be needed, and avoids executing speculative tasks that will never be requested. Specifically, this section compares SRPT × HRP against SRPT × RFCFS. In addition, this section compares the non-size-based batchactive policy FCFS × HRP against SRPT × RFCFS. Mean scaled billed resources is omitted because it is always 1 for batchactive usage of a batchactive scheduler.

The improvement factors of mean visible response time is shown in Figure 6.62. Similar 's'-shaped improvements, which indicate no significant difference, were found for mean visible slowdown and requested load (not shown). Varying a single parameter below provides some justification for this result.

I varied the number of users, holding other parameters constant, to show its effect on visible response time in Figure 6.63. At a medium number of

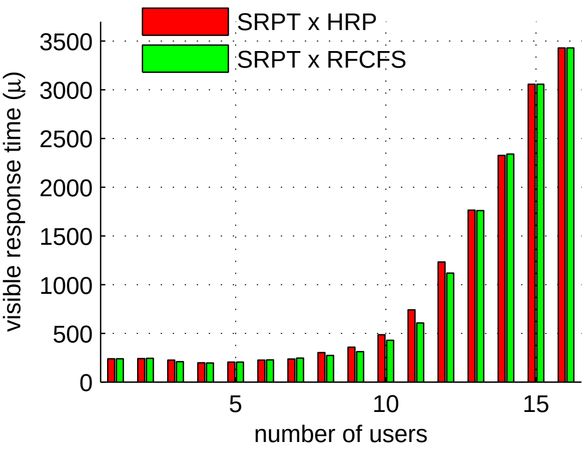


Figure 6.63: The effect of the number of users on batchactive usage of SRPT × HRP and batchactive usage of SRPT × RFCFS for mean visible response time. Each set of scheduling configurations was run with a different number of users simultaneously competing for the shared resource while other user and task characteristics were held constant. Since lower visible response time is better, bars of less height indicate better performance. SRPT × RFCFS *provides slightly better performance at a medium number of users.*

users, SRPT × RFCFS provides slightly better mean visible response time by ignoring disclosed tasks that will never be requested. At the extremes of numbers of users, performance is the same — either there is not enough disclosed work for the disclosed queue subpolicy to make a difference (low numbers of users), or there is so much needed work that the disclosed queue is never serviced (high numbers of users).

Other metrics and varying other parameters did not show significant differences between the policies. Only extremely speculative task sets (not shown and not included in the inverse cumulative improvement data in Figure 6.62) display a marked benefit for SRPT × RFCFS.

In addition, I compared the implementable batchactive policy that does not require size information, FCFS × HRP, against the oracle batchactive policy, SRPT × RFCFS. The improvement factors of mean visible response time is shown in Figure 6.64. The SRPT × RFCFS policy, which is impractical on its reliance of knowing task size and avoiding unneeded speculation, performs better than FCFS × HRP. Favoring small requested tasks affects visible response time to a greater extent than the choice of disclosed queue subpolicy.

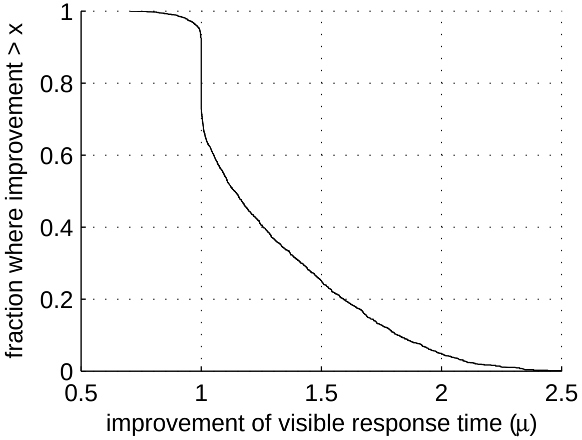


Figure 6.64: Improvement of SRPT × RFCFS over FCFS × HRP for mean visible response time. A task's visible response time is the time between a user needing and receiving the task's output. The horizontal axis shows improvement factors while the vertical axis indicates the fraction of the runs in which the improvement was *at least* as much indicated on the horizontal axis. *The area below the curve for improvements greater than* 1 *show that* SRPT × RFCFS *performs better than* FCFS × HRP. The arithmetic mean improvement is 1.284 and the geometric mean improvement is 1.244.

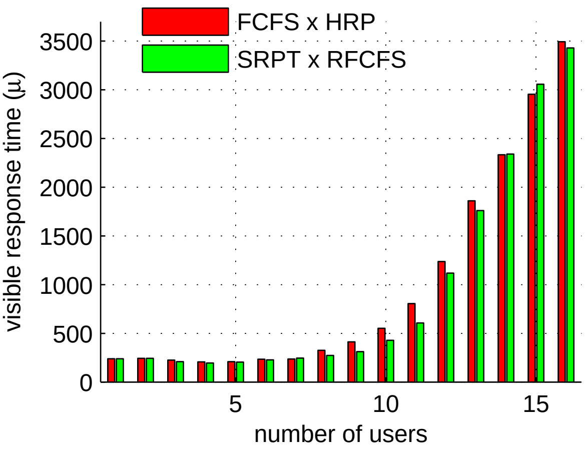


Figure 6.65: The effect of the number of users on batchactive usage of FCFS × HRP and batchactive usage of SRPT × RFCFS for mean visible response time. Each set of scheduling configurations was run with a different number of users simultaneously competing for the shared resource while other user and task characteristics were held constant. Since lower visible response time is better, bars of less height indicate better performance. SRPT × RFCFS *provides slightly better performance at a medium number of users.*

I varied the number of users, holding other parameters constant, to show its effect on visible response time in Figure 6.65. This data does not show much difference between SRPT × RFCFS and FCFS × HRP. The difference between these two schedulers is not illuminated by varying the number of users. Instead, the performance difference in Figure 6.64 results from SRPT handling larger tasks better than FCFS for requested tasks. This could be shown by varying the service time parameter.

This section showed that HRP as the disclosed queue subpolicy is competitive with an impractical disclosed queue subpolicy. Further, this section showed that size-unaware batchactive scheduling using an implementable disclosed queue subpolicy performs worse than the oracle batchactive policy. However, the limit of batchactive performance is unknown; SRPT × RFCFS does not take into account task size for speculative tasks nor does it know when a task will be requested, i.e., task deadlines. As discussed (Chapter 5.5.3), predicting deadlines, even in simulation, is difficult.

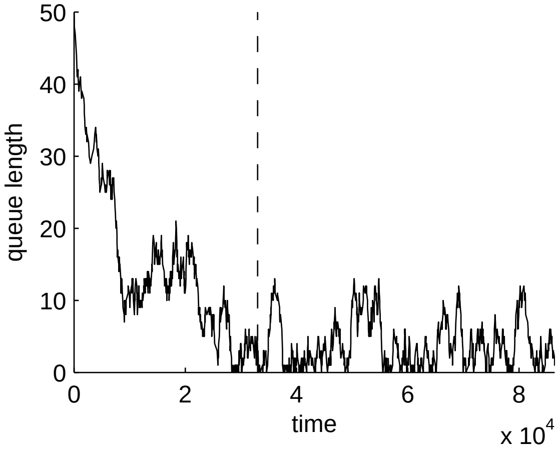


Figure 6.66: The queue length of requested tasks for an extreme selection of simulation parameters stabilizes after approximately 10 hours of simulated time. (The numbers on the horizontal axis are seconds times 10^4.) For this reason, all reported metrics avoid including such warm up data by conservatively dropping the first two days of simulated time.

6.3 Simulation details

Here I state several simulation details. First I describe how the simulation was run to eliminate non-steady-state behavior. Then I present some confidence interval data to show that the differences in metrics across compared runs are significant. Finally, I describe some aspects of the simulator runs including the kinds of machines used, how many simulations were needed for this research, and how long they took to run.

6.3.1 Omitted warmup period

When the simulation starts, all users begin submitting tasks. Only after task outputs are received do users enter their think times. Thus the queue lengths near the start of the simulation are not representative of the behavior of the system at steady state. The most extreme example that I found is illustrated in Figure 6.66. I avoid including such warmup data (also called ramp-up or run-in data) in the reported metrics by conservatively dropping all data in the first two days of simulation time. Dropping warmup data to focus on steady-state behavior is common in discrete event simulations [Ball, 2004]. By dropping two warm up days from 16 day simulations, the presented metrics reflect two weeks at steady state.

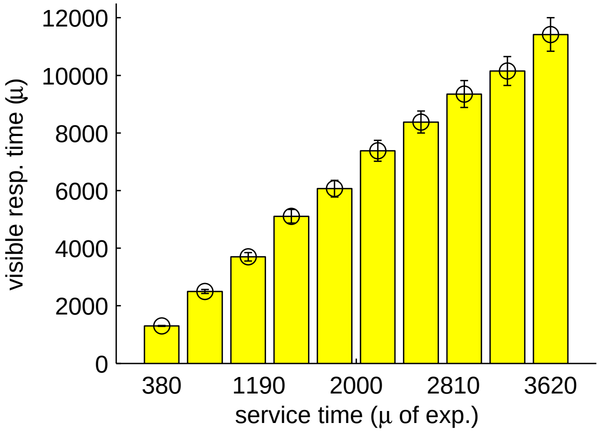


Figure 6.67: Confidence intervals for a small run show that the results are significant. Plotted is the mean visible response time across ten service times. The 95% confidence intervals were from 40 runs using different random seeds at each service time.

6.3.2 Statistical significance of the results

The experiments in Chapter 6.2 show that the simulation parameters effect scheduling metrics. To reinforce this observation, I took confidence intervals of a small run to measure the significance of the tabulated metrics. This aspect of model validation (Chapter 6.1.4) is called *internal validity* [Sargent, 1999].

Figure 6.67 shows the mean visible response time for a small interactive SRPT run in which service time was varied in six minute increments while other parameters were held constant. Each reported visible response time mean and confidence interval was the result of 40 simulations that were started with different random seeds. Out of all selections of service times, only one pair of 95% confidence intervals overlapped, and all confidence intervals were less than 5% of their respective mean visible response times. A normal probability plot (not shown) of the response times for each service time revealed that they were sufficiently normally distributed for the confidence intervals to be reliable. Thus the simulation results, which are a function of simulation parameter selections, are statistically significant.

6.3.3 An accounting of the simulator runs

Roughly 100,000 runs were performed on a Condor [2003] cluster of ten 2.4 GHz Pentium IV machines each with 1 GB of memory shared with other graduate students. Most runs took less than ten seconds and less than 100 MB of memory to simulate 16 days of simulated time of which data from the initial two warmup days were not included in the presented results (Chapter 6.3.1). A small percentage of runs with parameters causing many more tasks to be created took roughly ten minutes to run.

I desired a batchactive scheduler while exploring the simulator's large parameter space. The service and think time of this research were ideal for batchactive scheduling and fit the parameter study application scenario described in Chapter 2.2.3.

6.4 Summary

This chapter presented the quantitative results of this thesis showing that better time- and cost-based scheduling metrics are achievable with batchactive schedulers compared to non-speculative schedulers.

The results were obtained through two-week simulations. The simulator realizes a model of user behavior, task characteristics, and a single server. When using a non-speculative scheduler, simulated users either behaved interactively or in a batch manner. The simulations of non-speculative schedulers had all interactive or all batch users. When using a batchactive scheduler, simulated users behaved in a batchactive manner, disclosing speculative tasks. The simulated model, motivated by real application scenarios, determined the deadlines of speculative tasks. Validation and verification techniques were applied to provide confidence in the simulation results.

A number of parameters were needed to realize this model. The simulator can vary the number of users, the probability that a task set will be canceled after a user considers the last output received, the number of tasks per task set, the service time distribution, and the think time distribution.

The experiments began with a comparison of the simplest non-speculative and speculative schedulers: FCFS and FCFS × FCFS. Other experiments examined novel disclosed queue subpolicies (including an impractical disclosed queue subpolicy), usage-based scheduling, and size-based scheduling. The main metrics tabulated were mean visible response time, mean visible slowdown, mean scaled billed resources, and requested load. The first two are time-based and the last two are cost-based.

Under a variety of user and task behavior, batchactive usage of a batchactive scheduler nearly always does better than convention, in which tasks are requested one at a time (interactively) or requested in batches to non-speculative schedulers which cannot discriminate between speculative and non-speculative tasks. With a batchactive system, users do not need to decide how aggressively to submit speculative work.

Batchactive scheduling is adaptive: it is at least as good as interactive or batch usage of non-speculative policies as the number of users varies between extremes, and in the middle ranges it is better than both. The batchactive approach applies best when several to many speculative tasks are submitted and early task outputs are acted on while uncompleted tasks remain.

The disclosed queue must be scheduled carefully to avoid a diminishing returns when users disclose a large amount of unneeded work. A disclosed queue subpolicy which learns from past user behavior, HRP, schedules disclosed tasks well, comparably to an impractical disclosed queue subpolicy.

Latency-sensitive users will not push traditional schedulers into regions of high billed load because, at those levels of revenue, visible response times are too high. The significant value provided with batchactive scheduling with respect to time-based metrics could encourage additional users, deeper speculation, and bigger tasks, any of which would raise batchactive server revenue.

I ended this chapter with some details of the simulation including the omission of warmup periods, the statistical significance of results, and an accounting of several aspects of the simulation runs.

Look upon my works, ye Mighty, and despair!
Percy Bysshe Shelley, *Ozymandias*

7 Implementation & proposed deployment

This chapter details the simulator's features, structure, coding practices, and scheduling overhead and describes how batchactive scheduling may be deployed as an extension to existing clustering software.

7.1 The ba_sim simulator

The largest artifact I created in this work is the ba_sim discrete event simulator used to test my thesis. A discrete event simulation is a common way to implement models to observe time-based (dynamic) behavior [Ball, 2004]. Results from ba_sim were presented in Chapter 6.2. I contribute ba_sim to the research community. It may be downloaded from http://www.pdl.cmu.edu/PDL-FTP/Scheduling/ba_sim-0.1.tar.gz. The file named README in this archive describes how to use the simulator. This section describes the features, structure, coding practices, and scheduling overhead of ba_sim.

7.1.1 Features

The ba_sim simulator implements the user, task, and server model described in Chapter 6.1. The inputs and outputs to the simulator are depicted in Figure 7.1.

There are two modes in which users may arrive in ba_sim. In one, users arrive according to a Poisson process and depart after one task set is exhausted or canceled. Paxson and Floyd [1994] show that while some activity should not be modeled by a Poisson process, like the packets in an ftp transfer, user-initiated events, like making a telnet connection, or in my case, a user arriving, can indeed be modeled well by a Poisson process. Results from this thesis do not come from runs in which users arrive and depart; instead, ba_sim was in a mode in which a constant number of users arrive at the start of a run and never depart, issuing new task sets when needed. Small runs of ba_sim in the mode in which users may arrive and depart show batchactive

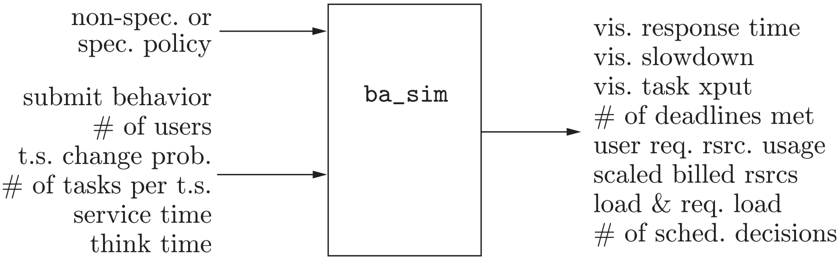


Figure 7.1: Inputs (from left) and outputs (to right) of the `ba_sim` simulator. Scheduling policy, user characteristics, and task characteristics determine scheduling metrics. The policies can be speculative or non-speculative (e.g., FCFS × FCFS or FCFS, respectively). User and task characteristics include whether the users are batch, interactive, or batchactive in the way they submit task sets, the number of users simulated, the task set change probability, the number of tasks per task set, task service time, and user think time. Submit behavior was described in Chapter 6.1.2 and task characteristics and other user characteristics were described in Chapter 6.1.3. Output metrics include visible response time, visible slowdown, visible task throughput, the number of deadlines met, user requested resource usage, scaled billed resources, load, requested load, and the number of scheduler decisions. Metrics were described in Chapter 5.2.

improvements under the same situations discussed in Chapter 6.2.2.

The `ba_sim` simulator can also take some inputs from traces instead of from random variables, such as when users arrive and the service times of tasks. This trace mode was not used to generate results because of the difficulty in arguing for representative traces and because the community generally accepts the random variables that I used to model user and task characteristics.

Besides terminating a simulation after a certain amount of simulated time, `ba_sim` may also be instructed to terminate after a certain number of users arrive (in the mode in which users arrive according to a Poisson process) or after a certain number of tasks finish. All results in this thesis are from runs which terminate after a certain number of time to simplify the comparison of metrics between configurations. The downside to this approach is that statistics from one configuration, such as mean visible response time, may be computed from a different number of samples than another configuration.

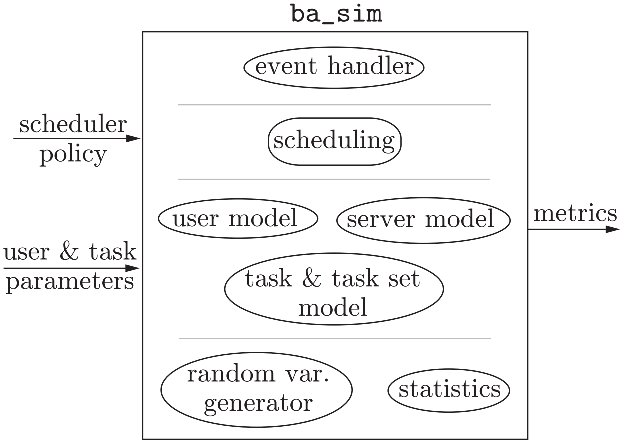


Figure 7.2: Structure of the ba_sim simulator. Scheduling policy, user parameters, and task parameters are input to each run of the simulator. Scheduling metrics are output. The simulator consists of an event handler to track discrete events such as a preemption or a user's think time elapsing. The scheduling module implements scheduling policies. The user, server, task and task set modules implement the model described in Chapter 6.1. Helper modules generate random variables and track statistics. Minor helper modules are omitted from this representation.

7.1.2 Structure

I wrote ba_sim in C for the Darwin (Mac OS X) and Linux platforms. The build system uses automake and autoconf to simplify building and porting. The ba_sim simulator consists of modules that simulate user, task, and server behavior as shown in Figure 7.2.

The simulation runs were handled by helper scripts. One Perl script called harness executes large numbers of instantiations of ba_sim to explore its parameter space and stores simulation output in a MySQL relational database. I use Condor [2003] to distribute copies of harness across a cluster. Database accesses check the existence of specific completed runs so that many harness copies can cooperatively run ba_sim configurations with little wasted work. Output metrics are atomically inserted into the database to ensure no corruption if two or more harness copies try to insert data from the same ba_sim configuration.

The improvement Perl script calculates factors of improvement among scheduling configurations. The present Perl script uses improvements and raw ba_sim metrics to generate graph files and summary output. Graph files are delivered to the MATLAB technical computing system for visualization.

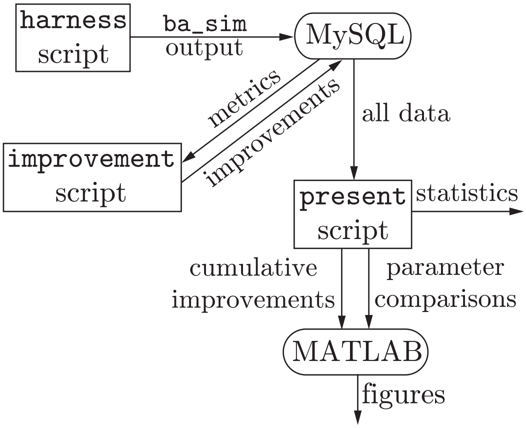


Figure 7.3: The interaction between ba_sim and the tools used to generate thesis results (Chapter 6.2). The harness script is distributed to a cluster using the Condor [2003] clustering system. Each copy of harness executes numbers of ba_sim instances, collects ba_sim output, and inserts them into a MySQL database. The improvement script calculates improvement factors among configurations by querying ba_sim outputs from the database and inserting improvements into the database. Raw metrics and improvements are requested from the present script to generate graph files delivered to MATLAB to produce figures and summary statistics used in tables and figure captions. Some scripts are not shown (such as to administer the database and produce some statistics).

This toolchain around the ba_sim simulator is depicted in Figure 7.3.

The number of lines of code (including comments) for ba_sim is 14, 542. The number of lines of code for Perl scripts and Perl modules shared by some scripts is 9, 115. There are also Condor description files for distributing copies of harness totaling 1, 329 lines and bash shell scripts for automating the executing of the improvement and present scripts totaling 6, 602 lines.

7.1.3 Coding practices

I applied well-known software engineering techniques to give me confidence that the results conform to my model of users, tasks, and one server; that the results generated by ba_sim are accurate (Chapter 6.1).

The ba_sim simulator code contains over 500 checks ('assertions') which increase confidence in the integrity of simulation results. I calculate scheduling metrics in several ways when possible and ensure their results match. When I am aware of an invariant, I verify it; e.g., if the summation of some metrics cannot exceed a value, I assert that this is the case.

FCFS, batch	FCFS, interactive	FCFS × FCFS
2.524 (0.468)	1.055 (0.024)	2.246 (0.197)

Table 7.1: Total time in milliseconds averaged over 35 runs started with different random seeds to perform scheduling decisions over two weeks of simulated time. The 95% confidence interval of each mean is the mean plus and minus the value in parenthesis. 3 ms over two weeks is insignificant and the overhead for FCFS × FCFS decisions is between that for interaction and batch usage of FCFS. The number of finished tasks for all three configurations ranged from 1, 200 to 1, 500. The number of scheduler decisions for all ranged from 2, 000 to 2, 700 (see Table 6.6).

I carefully monitored debugging output, conducted redundant detailed and high-level statistical comparisons, hand-inspected degenerate cases, and hand-inspected specific cases. Moreover, constants were used for various input parameters so that I could check some results by hand.

These practices for simulator verification (Chapter 6.1.4) are an important subset of those advocated by Sargent [1999]. More verification would be possible in a real world test, which is outside my scope.

7.1.4 Overhead

I measure scheduling time overhead in `ba_sim` because it is important for a scheduler to not lose efficiency with excessive or costly decisions. The `ba_sim` simulator uses non-optimal data structures (cleverness took a backseat to correctness): the list data structures from the `glib` library of general data structures instead of priority queues which would reduce most scheduling operations from most policies from $O(n)$ to $O(\lg n)$ [Cormen et al., 1990]. Even with lists, I determine the overhead to be negligible.

Scheduling time is summarized in Table 7.1. These numbers were taken on a 1.25 GHz PowerPC G4 processor. The system was near idle, but undoubtedly background load occasionally inflated some timings. Most time in my simulation concerns the accounting used to generate the metrics used in comparing scheduling policies and to verify their correctness. This accounting would not be activated in a real implementation, and thus is not tracked.

7.2 Cluster scheduling extension

Batchactive scheduling may be deployed as an extension to a clustering system. Many tasks from the same user can execute in parallel on a cluster (Figure 2.1) if only those tasks were known to the scheduler. Speculative disclosure is discouraged on existing systems because no interface exists for informing the scheduler which tasks are speculative. If a user were to speculate, non-speculative tasks would starve behind speculative tasks. Further, speculative disclosure is discouraged on existing systems when resource usage is charged because users seek to avoid being charged for unneeded work (Chapter 5.1.5).

I first discuss user interface and coding considerations on clustering systems before discussing the extensibility of existing clustering systems. I concentrate on the extensibility of Condor [2003].

7.2.1 Usage of a clustering system

User interface considerations have been addressed in the context of non-speculative scheduling. Task control (requesting, disclosing, canceling) and task querying (tracking what is running, resource consumption, searching for tasks, etc.) can be done at the command-line [Condor, 2003] or through the web [NEOS, 2004]. Existing systems can notify the user of finished tasks by e-mail [Condor, 2003]. Task inputs and outputs may be located on a shared file system [Condor, 2003] or transmitted through the web [NEOS, 2004]. Dependencies among tasks can be specified using directed acyclic graphs [Condor DAGMan, 2004]. Coding guidelines and libraries for enabling tasks to migrate among nodes exist [Condor, 2003].

7.2.2 Extensibility of existing systems

Extending a clustering solution to support batchactive scheduling requires adding the ability for users to disclose tasks, modifying the scheduling policy, and isolating speculative task outputs until requested. Task output isolation was discussed in Chapter 5.6.6 and is not discussed further. A frontend that enables users to disclose or request tasks, instead of only request tasks, could reside between the user and any clustering system. This frontend could tag task type (request or disclosure) so that a modified scheduling policy can treat different types differently. Some systems enable library or module replacement of their stock schedulers. Others are extensible through parameters affecting scheduling decisions. Finally, some systems can only be extended through source code modification. For these, when source is not available, one could start with the ideas concerning non-invasive frontends described in Chapter 5.9.2.

Parsons and Sevcik [1997] discuss extensions to the LSF clustering system [Platform, 2003]. They found that building on this commercial system was straightforward. LSF provides an application-programming interface (API) enabling many scheduling aspects to be controlled, obviating the need to modify source code. I do not explore LSF further because its commercial nature makes it difficult for many to obtain.

Xgrid [2004] from Apple Computer, Inc. is proprietary, closed-source, and non-extensible. Its scheduling algorithm is simple: tasks are run on nodes in the order in which they are received, with individual tasks assigned to the fastest nodes first [Prabhakar, 2004]. The best approach to implementing batchactive scheduling on Xgrid might be with non-invasive frontends (Chapter 5.9.2).

7.2.3 Extending the Condor clustering system

The Condor [2003] clustering system simplifies the use of cluster resources by hiding the details of remote execution and providing task checkpointing facilities. Condor is believed to be the most popular clustering system. I describe how it can be extended, as alluded to in Chapter 5.6.3. These ideas were formed in consultation with the principal investigator of the Condor Project [Livny, 2004] who considers batchactive scheduling novel and a potential batchactive scheduling Condor extension useful.

I describe how to add the ability for users to disclose tasks and how to modify the Condor scheduling policy.

Task requests are specified in a `ClassAd` format which specifies task requirements as a set of extensible characteristics. For non-speculative tasks, the user may use the existing `condor_submit` command to request tasks.

To disclose tasks, the Condor DAGMan [2004] tool can be harnessed to execute a user's task set along with a Condor-specific batchactive user interface which manages the transition of a task from being disclosed to requested and the storage and delivery of task outputs to the user (Figure 7.4). Consider, for simplicity, list-ordered task sets. The batchactive interface follows this list, providing a prompt for the user to request the next task in the task set and holding outputs until needed. The user may request the next task output at any time. If the task has not executed, its priority is elevated to 'needed.' If the task has executed, its output is supplied to the user. Using Condor DAGMan provides a way to let task execution get arbitrarily ahead of user task output desire, while enabling a channel for the batchactive interface to track of task progress.[1]

[1] Note that DAGMan is employed to present the batchactive user interface; not necessarily to enable general task set orderings, although that would additionally be possible.

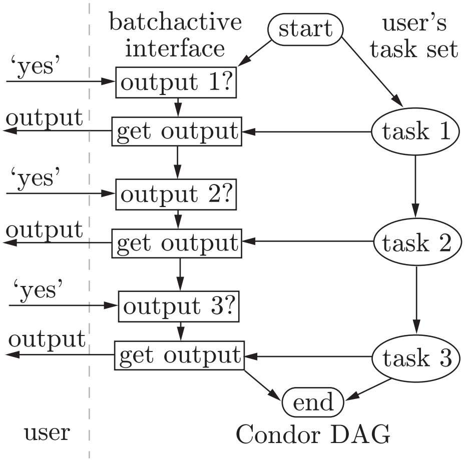


Figure 7.4: Proposed user interface batchactive extension to Condor [2003]. The vertical column of tasks on the right represents a single user's list-ordered task set. The left part of this directed acyclic graph is a Condor-specific batchactive interface which asks the user if the user needs particular task outputs and waits for outputs to be produced if not yet available. The arrows in the DAG represent dependencies: the user will not be prompted for tasks out of order. The current prompt appears immediately, before the task has begun executing, enabling a user to elevate a disclosed task to requested at any time. Since there are no dependencies in the other direction, from user interaction to tasks, task execution is permitted to get arbitrarily ahead of user interaction.

The priority of a task, a function of whether the task is speculative or needed, is controlled by the batchactive interface modifying an attribute in the task's `ClassAd`. The `ClassAd` attribute is examined by the scheduler (described below) to determine execution order. If a task has already begun, then to have the `ClassAd` honored for scheduling, the task will need to be stopped and restarted.[2]

Condor's scheduling strives for fair, decayed resource usage among users over long time periods (Chapter 4.6.2). This is accomplished with two types of priorities: user priorities and task ('job') priorities. The number of machines assigned to a user is inversely related to the ratio of user priorities among users; e.g., a user with priority 10 will get twice as many machines as a user with priority 20. Each user specifies task priorities for his or her own tasks. These priorities specify an absolute ranking among a user's tasks on the machines assigned to the user. Forty task priorities are available, limiting the prioritization that can occur among submitted tasks. Condor's scheduling policy is not extensible without modifying source. Although Condor is open-source, its source code has not been released by the time of this writing.

Condor scheduling is handled by several daemons, the `negotiator` (also called 'matchmaker') and `schedd`. Task requests go to each local `schedd` which places tasks into a queue. The `negotiator` matches task requests to machines, discovering queued tasks by querying `schedd` and heeding `ClassAd` requirements. The `negotiator` is responsible for enforcing user priorities (inter-user scheduling) and `schedd` is responsible for enforcing task priorities (intra-user scheduling).

The first-level user and second-level task scheduling of the `negotiator` and `schedd`, respectively, frustrates the ability to implement new, global scheduling policies. One way to circumvent Condor's decayed user-based resource usage scheduling is to submit all tasks from a single 'user' and leverage the `ClassAd` `rank` floating-point attribute. The task with the higher `rank` receives a machine before one with a lower `rank`.

The two-tiered batchactive scheduling FCFS × FCFS may be implemented with the following rank settings: all disclosed tasks are assigned a rank of 0.timestamp and requested tasks are assigned a rank of 1.timestamp, where 'timestamp' is the time that the task was either requested or disclosed. SRPT may be implemented by setting the rank to the inverse remaining service

[2] The benefit of such restarting should outweigh the overhead cost. One part of the overhead is whether completed work needs to be recomputed. With checkpointing (a feature of Condor) in effect, completed work would be preserved.

time. Two-tiered scheduling based on SRPT may be accomplished by multiplying requested tasks by a sufficiently large constant. Less efficiently, any scheduling order (including HRP and HRR) may be determined by calculating a monotonically increasing numerical ranking for each task and then writing the `ClassAd` for every task when the ranking changes (which may occur frequently).

Although the feasibility of this description has been vetted by a Condor expert, a prototype would likely uncover inaccuracies in details and illuminate needed revisions.

7.3 Summary

I detailed the `ba_sim` simulator used for my thesis results, listing its features, describing its structure, arguing that employed coding practices provide confidence in its results, and showing that scheduling overhead in its unoptimized implementation is negligible. I showed how batchactive scheduling can be deployed as an extension to popular clustering software. Existing cluster interfaces do not prevent deploying batchactive scheduling, but some systems are easier to extend than others. I described a way to deploy batchactive scheduling on the Condor clustering system.

From Rusticus I received the impression that my character required improvement and discipline; and from him I learned not to be led astray to sophistic emulation, nor to writing on speculative matters [...].

Marcus Aurelius, *The Meditations*

8 Conclusions

I presented a scheduler for clusters, grids, and supercomputers to assist users speculatively searching for interesting task outputs. Batchactive scheduling harnesses and bounds searches to maximize human productivity while minimizing unnecessary resource consumption, exploiting the trend that the cost of human time increases while the cost of computing time decreases.

In this chapter I restate the addressed problem and my primary contributions. I end with thoughts on overcoming non-technical challenges for the widespread acceptance of batchactive scheduling.

8.1 Problem restatement

Scientific disciplines and commercial ventures use computer resources to simulate phenomena, evaluate hypotheses, visualize information, discover invariants. Individuals often construct series of experiments occupying considerable computing time, in which, at the outset, it is often unclear which task outputs will be useful. This speculative behavior motivates my thesis.

A speculative task is some unit of work, usually corresponding to a run of an application, whose output is not yet known to be required. I call all the speculative tasks associated with a user his or her task set. Exploratory searches, sequential tasks, and parameter studies are common types of speculation formed of task sets. An exploratory search is typically a hand-crafted chain of speculative tasks from different applications whose outputs increasingly provide evidence to confirm or refute a hypothesis. A search comprised of sequential tasks is one in which all tasks are from a single application such as an any-time algorithm providing increasingly detailed output or ordered, temporal outputs. A parameter study is a set of tasks exploring a large parameter space, usually beginning by exploring the space in broad, shallow strokes, later to be refined to specific areas of interest. Examples for each are issued by bioinformaticists comparing DNA sequences, computer graphics artists rendering scenes, and computer researchers studying cache behavior, respectively. These examples are important both in the economic sense and in the advancement of science.

Users often plan ahead, wishing to pipeline the consideration of received task outputs with the execution of speculative tasks whose outputs were not known to be needed at the time of submission. Ideally, a cluster task scheduler would run speculative tasks while users were analyzing completed tasks, minimizing the users' waiting time. The catch is that speculative tasks will take contended resources from users who are waiting for known-needed, non-speculative tasks unless the two types of tasks can be discriminated. Being too far ahead takes contended resources away from others. This is particularly inefficient when outputs from early experiments would render the execution of the long-range, speculative experiments unnecessary.

Existing cluster scheduling and pricing mechanisms are not designed with speculative tasks in mind. This causes confusion on the user's part and suboptimal scheduling. Should a user exploring a space speculatively submit one speculative task, a few, many, or the entire 'computational plan?' A cost-aware user will submit a few or no speculative tasks to avoid being charged for unnecessary speculation. Such a user will experience poor visible response time because there is no opportunity for the system to execute potentially needed tasks while the user considers received task output. A user with the means to pay for speculation or using resources that are not directly charged will speculate maximally in an attempt to minimize visible response time. When many behave this way, resources are swamped by deep speculation and needed tasks are starved behind many tasks that will turn out to not be needed. Further, not enough information is available to each user, even if he or she was willing to take the burden, to decide an optimal number of speculative tasks to submit. The right number would depend on other task arrivals, task service time, user think time, and the probabilities that speculative tasks will be needed.

Non-speculative schedulers are unaware of which tasks are speculative and thus do not treat speculative tasks differently from non-speculative tasks. When a user speculatively issues tasks, there is a notion of task deadlines; the time that a speculative task might be needed which occurs later due to user think time. Think time is not exploited by traditional schedulers to favor more pressing tasks. Further, the interface presented by traditional schedulers makes it difficult to expose user think time to the scheduler. Finally, non-speculative schedulers are unaware which speculative tasks have turned out to be needed, or which users speculative less than others, and thus non-speculative schedulers are not able to favor tasks that are more likely to be needed.

The existence of speculation uncovers new questions for scheduling theory and practice: how should a policy order speculative, abortable tasks to minimize the time users are blocked on task output, minimize user costs, and maximize server revenue, and what interface should a scheduler present to the user to accomplish these goals?

8.2 Primary contributions

I introduced batchactive scheduling to alleviate the problems presented by traditional, non-speculative scheduling. The solution I promote exploits the inherent speculation in common application-level searches. I focused on non-parallel applications, leaving the speculative scheduling of parallel applications, which require coscheduling and the prediction of multiple resource availability, to future work.

Batchactive scheduling consists of several ideas: speculative tasks are treated differently from non-speculative tasks; the time-based metric that is minimized is visible response time; and users are encouraged to speculative deeply with the batchactive pricing mechanism. These ideas are enabled by a batchactive interface.

With batchactive schedulers, users disclose speculative tasks and request tasks whose outputs they know they need. (With a non-speculative scheduler, a user would have to throttle his or her own speculation, which is difficult and burdensome, in an attempt to meet time and cost goals.) A user may cancel any task if received outputs suggest their irrelevance. I call this 'batchactive' usage of the system, because, like batch usage, many tasks are submitted at once, and, like interactive usage, the user is waiting for the output of (usually) one identified task.

Disclosure enables the scheduler to get an early start, fulfilling user desire to pipeline think time and task execution. Further, disclosure exposes parallelism within a user's workload, enabling the scheduler to exploit the parallelism of cluster nodes.

I observe that not all tasks are equal — only tasks blocking users matter — leading me to introduce the visible response time metric which measures the time between needing and receiving task output, independent of when it was speculatively disclosed. With the request and disclose batchactive interface commands, visible response time is exposed to the system. Traditional response time, the time-based scheduling metric most often employed, conflates the time of task submission with the time of task desire. When a user is able to plan ahead, and when a user undergoes think times each time he or she receives task output, the time of task submission and desire are often not the same.

A user would prefer not being charged for disclosed but never requested tasks. Moreover, the more knowledge that the scheduler has of a user's computational plan, the better it can order tasks to meet important scheduling goals. To motivate users to disclose freely and deeply for tasks likely and unlikely to be needed, only requested resources are billed. This is the batchactive pricing mechanism. Disclosed tasks that were never needed are not charged. The billing system is able to implement this pricing mechanism in part because the batchactive interface provides separate request and disclose commands.

The batchactive interface also employs an isolated output store. If a task executes and was disclosed but not requested, then the task's output is stored in a location isolated from the rest of the system until requested or canceled. If canceled, its output is removed from the system. When the task is requested, the batchactive system provides the task's output to the requesting user.

This output isolation supports two aspects of batchactive scheduling: Output isolation enables the batchactive pricing mechanism, for without it, a user may attempt to read needed output that was speculatively generated without requesting and being charged for the output.

Moreover, output isolation forces the user to request needed task output and to cancel unneeded task output. Doing so provides useful information to the scheduler. Ambitious policies that learn from historical user patterns become possible. The traditional scheduling interface, without output isolation, will miss task requests or cancelations if a task executed while a user was still in his or her think time, because, in this case, there is no motivation for the user to explicitly request or cancel the task.

In simulation, I demonstrated my thesis that a multiuser process scheduler informed of which submitted tasks are speculative can provide better time- and cost-based metrics for users and resource providers. Simulation runs compare batchactive scheduling to interactive and batch usage of a non-speculative scheduler in which users request needed tasks one at a time or in non-speculative batches, respectively. The simulator I created for these experiments is called `ba_sim` and is available at `http://www.pdl.cmu.edu/PDL-FTP/Scheduling/ba_sim-0.1.tar.gz` for further research by others.

The simulated batchactive schedulers give requested tasks precedence over disclosed tasks. I call these 'two-tiered batchactive schedulers.' Deferring the execution of speculative tasks can save work when such tasks are later canceled; i.e., deferred work is often saved work. Simulated requested queue subpolicies included first-come-first-serve (FCFS), shortest-remaining-processing-time (SRPT), and a policy which selects the user who has used the least requested resources (user-requested-FB). Simulated disclosed queue subpolicies included FCFS, SRPT, and two novel subpolicies: highest-request-probability (HRP), which favors users who have historically requested a greater fraction of disclosed tasks, and highest-requested-resources (HRR), which favors users who have requested the most resources. Both HRP and HRR would not have been possible without output isolation to force users to indicate task output desire. In addition, I explored the performance of an impractical, oracular disclosed queue subpolicy which runs only disclosed tasks that will eventually be requested (RFCFS). I notate two-tiered batchactive schedulers as 'requested task subpolicy' × 'disclosed task subpolicy.' The simulated non-speculative schedulers included FCFS, SRPT, and a policy which selects the user who has used the least resources (user-FB).

For each scheduling comparison, the simulator was run thousands of times with different parameters describing user behavior and task characteristics. I chose parameter ranges that not only included reasonable uses of speculation for my target applications, but also ranges that included little or no speculation and little think time, regions where little or no batchactive improvement was expected.

The time- and cost-based scheduling metrics output by the simulation runs support batchactive scheduling. Over a wide variety of runs, for mean visible response time, FCFS × FCFS performed at least twice as well for about 15% and 25% of the simulated behaviors of batch usage of FCFS and interactive usage of FCFS, respectively. When looking at mean visible slowdown (recall that visible slowdown is visible response time scaled by task size), FCFS × FCFS performed at least twice as well for about 25% and 30% of the simulated behaviors of batch FCFS and interactive FCFS, respectively. Size-based schedulers based on SRPT provided comparable relative batchactive improvement. Further, FCFS × FCFS performed better than size-based non-speculative scheduling. Thus the availability of a task size oracle will not diminish the value of batchactive scheduling. In all of these results, visible throughput (the throughput of needed tasks) was at least as good and often better under batchactive scheduling.

Batchactive usage of a speculative scheduler adapts across a range of task and user characteristics, often beating any usage of a non-speculative scheduler. For example, batch usage of a non-speculative scheduler is suited to few users because execution time and think time are pipelined and load is sufficiently low that one's speculative but unneeded tasks do not overly interfere with needed tasks. Interactive usage of a non-speculative scheduler is suited to many users because the server is always busy with requested tasks. Batchactive scheduling is better than batch usage of a non-speculative scheduler under many users because requested tasks never wait for speculative tasks; it is better than interactive non-speculative scheduling under few users because it fills idle time with speculative tasks.

In practice, besides FCFS, a variant of decay-usage is sometimes employed. Batchactive scheduling, such as user-requested-FB × HRP or FCFS × HRP, performed better than batch usage of user-FB. As the number of users are increased (which increases load), the batchactive configuration performed roughly 20% better. This is because the batchactive case executes known-needed tasks first while the user-FB policy might execute disclosed tasks that will be canceled even when known-needed tasks exist.

By varying task set size, the likelihood that a task output will cause a user to cancel remaining tasks, and user think time, I showed that batchactive scheduling applies best when several to many speculative tasks are submitted and early task outputs are acted on while uncompleted tasks remain. Think time is needed to obtain time-based benefits; without it, all scheduling configurations perform the same. With think time, the greater the degree of speculation, the better batchactive scheduling performs relative to non-speculative scheduling. As the number of tasks in the average task set increases beyond 10, the mean visible response time of batch usage of FCFS became unusable. Throughout a range of average tasks per task set from 10 to 512, the improvement of batchactive FCFS × HRP over interactive FCFS was between a factor of 1.5 and 3. As the cancelation probability for the average task set increased from no speculation to 0.2, batchactive improvement over batch ranged between a factor of 2 and 6.

The HRP disclosed queue subpolicy was shown to be better than using FCFS for the disclosed queue as the number of tasks per task set was increased. When the average task set had roughly 50 tasks, mean visible response time under FCFS × HRP was roughly half than under FCFS × FCFS. Thus, the disclosed queue must be scheduled carefully to avoid a diminishing returns of batchactive improvement as the disclosed queue fills with tasks less likely to be requested. The HRP subpolicy also avoids a denial-of-service of the disclosed queue from users who disclose but never request tasks.

Regarding cost, simulations showed that batch users on non-speculative schedulers sometimes pay greatly for unneeded speculation. The average per-user billed over needed resources for batch FCFS rose to 12 as the average task set sizes increased. Interestingly, with better scheduling, this phenomenon became worse. Under batch SRPT, this mean scaled billed resources rose to 20. The potential fear of paying for unneeded speculation motivates the batchactive pricing mechanism. With this mechanism, the mean scaled billed resources is always 1 because unneeded speculation is not charged.

The batchactive pricing mechanism is beneficial from the user's perspective because speculative work can be submitted at no cost. It can even work for the resource provider. Interactive usage of a non-speculative scheduler was shown to earn the resource provider the least revenue because these users only request one task at time. A batchactive scheduler often provided more total billed resources over the same time period in comparison, because, although in both situations only requested resources are charged, the batchactive case provided better visible task throughput. For example, at medium load, the batchactive case provided roughly 10% more billed load.

On the other hand, batch usage of a non-speculative scheduler was shown to earn the resource provider the most revenue because users requested speculative work and all executed work, whether needed or not, was charged. At a low to medium number of users, the difference between the two was also roughly 10%. When varying all parameters (including those describing the degree of speculation), the resource provider on a batchactive system would need to price requested resources roughly 5% more to meet the revenue of a non-speculative scheduler used by batch users. Note that the apparent additional billed load from the traditional pricing mechanism for batch users may not occur in practice: batch users would likely resist deep disclosure because they do not wish to be charged for needless speculation, and even if they were willing to pay, the resulting visible response times would be poor. Latency-sensitive users will not push traditional schedulers into regions of high billed load because, at those levels of revenue, visible response times are too high. The latency threshold for batchactive scheduling, in contrast, is better. For example, at a mean visible response time of 1000 s, the requested (billed) load of the batchactive FCFS × FCFS case is 0.92 (out of a maximum of 1) while the requested load of the batch FCFS case is only 0.75.

In a cost-center, if the batchactive pricing mechanism results in lower revenue, the price of requests can be raised to make up the difference. Both interactive and batch users would receive better value in the form of the lower mean visible response times described above while the billing total would remain unchanged. A profit-center can be motivated to use the batchactive pricing mechanism because the significant value provided with batchactive scheduling could encourage additional users, deeper speculation, and bigger tasks, any of which would raise server revenue.

8.3 Challenges to acceptance

I demonstrated that, compared to non-speculative scheduling, batchactive schedulers reduce visible response time and increase the fraction of time that cluster resources spend on needed work. Speculative task disclosure and the batchactive pricing mechanism support how people wish to work for the target application scenarios. I determined batchactive scheduling overhead to be negligible and I described how batchactive scheduling can be deployed by extending the popular clustering software system called Condor.

Given the benefits of batchactive scheduling I have presented, one may wonder why something similar is not already widely deployed. I believe that the enabling technologies have only recently been in place and that there is an initial learning curve to be surmounted. First, the benefits of speculation require that not all resources are consumed by requested tasks; this is increasingly true in cluster and grid environments. Second, users must be able to anticipate, with a minimum degree of accuracy, tasks that they might need. These points are related: the better users separate disclosures from requests, the more resources are available to a batchactive scheduler.

Although effort is involved, I believe it is possible for users to convey speculation by first disclosing tasks. For my example applications, listing speculative tasks is nearly automatic: the consecutive frames in a scene to be rendered, the evenly-spaced data points in a high-dimension parameter space. A user who underreports potentially needed work will not realize the best performance, but will not hurt the performance of others. A user who overreports speculative work can only improve his or her own metrics (so long as unneeded work is canceled). With a suitable disclosed queue subpolicy such as HRP, the performance of others will not suffer from such overreporting.

Specifying useful task granularities is also important. Consider an application whose output must be processed with a tool before the output is useful. If the application and tool are considered separate tasks, then the user will have to disclose and request them independently. At best, this is burdensome. At worst, the user forgets to request the tool's output until after the application finishes, wasting time. Moreover, the user would be alerted by the completion of the application, which is distracting when only the tool's output can be used. On the other hand, if a task consists of both speculative and non-speculative work, then its granularity is too big, resulting in worse cost-based metrics for the user who does not eventually need all the task's output and worse time-based metrics for others whose needed tasks are queued behind this large task.

In the surveys that I conducted to learn how people use computers speculatively, I asked whether a batchactive scheduler would be useful to them. Answers were positive but cautious: because people do not have the ability to express speculation to existing clustering software, they do not have the mindset of deeply disclosing their computational plans. Doing so with existing systems is not free in the economic and latency sense: users would be charged more and be delayed by speculation issued by others. Surveyed users restrict their speculation in practice due to those costs. Only real world testing can determine whether users would exploit a task disclosure interface, even one as syntactically similar to the existing request interface, to obtain the benefits demonstrated by this thesis from better scheduling increasingly important, expensive searches in the advancement of science.

. . .

Bibliography

ABDELZAHER, T. F. 2000. An automated profiling subsystem for QoS-aware services. In *Proceedings of the 6th IEEE Real-Time Technology and Applications Symposium (RTAS '00)*. Washington D.C. 72

ABELSON, H. AND SUSSMAN, G. J. 1996. *Structure and Interpretation of Computer Programs*, 2nd ed. The MIT Press. 31

ABENI, L., PALOPOLI, L., AND BUTTAZZO, G. 2000. On adaptive control techniques in real-time resource allocation. In *Proceedings of the 12th IEEE Euromicro Conference on Real-Time Systems*. Stockholm, Sweden. 126

ACHARYA, A., EDJLALI, G., AND SALTZ, J. 1997. The utility of exploiting idle workstations for parallel computation. In *Proceedings of ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '97)*. Seattle, WA. 28, 118

ALTSCHUL, S., GISH, W., MILLER, W., MYERS, E., AND LIPMAN, D. 1990. Basic local alignment search tool. *Journal of Molecular Biology 215*, 403–10. 20

AMIRI, K., PETROU, D., GANGER, G. R., AND GIBSON, G. A. 2000. Dynamic function placement for data-intensive cluster computing. In *Proceedings of the USENIX 2000 Annual Technical Conference*. San Diego, CA, 307–322. 38

ANDERSON, E., BAI, Z., BISCHOF, C., BLACKFORD, S., DEMMEL, J., DONGARRA, J., CROZ, J. D., GREENBAUM, A., HAMMARLING, S., MCKENNEY, A., AND SORENSEN, D. 1999. LAPACK *Users' Guide*, 3rd ed. Society for Industrial and Applied Mathematics. 70

ANDERSON, T. E., CULLER, D. E., PATTERSON, D. A., AND THE NOW TEAM. 1995. A case for NOW (networks of workstations). *IEEE Micro 15,* 1 (Feb.), 54–64. 42, 44

ARANYA, A., WRIGHT, C. P., AND ZADOK, E. 2004. Tracefs: A file system to trace them all. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST '04).* San Francisco, CA. 115

Argonne 2004. Grand challenge applications — Argonne National Laboratories. http://www-fp.mcs.anl.gov/grand-challenges/. 17

ARLITT, M. AND WILLIAMSON, C. 1996. Web server workload characterization: The search for invariants. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '96).* Philadelphia, PA, 126–137. 24

ARPACI, R. H., DUSSEAU, A. C., VAHDAT, A. M., LIU, L. T., ANDERSON, T. E., AND PATTERSON, D. A. 1995. The interaction of parallel and sequential workloads on a Network of Workstations. In *Proceedings of the ACM Joint International Conference on Measurement and Modeling of Computer Systems (Sigmetrics '95 / Performance '95).* Ottawa, Canada. 28, 66

ARPACI-DUSSEAU, A. C. AND CULLER, D. E. 1997. Extending proportional-share scheduling to a network of workstations. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '97).* 111

BALL, P. 2004. Introduction to discrete event simulation. http://www.dmem.strath.ac.uk/~pball/simulation/simulate.html. 23, 131, 221, 225

BANSAL, N. AND DHAMDHERE, K. 2003. Minimizing weighted flow time. In *Proceedings of the 14th ACM-SIAM Symposium on Discrete Algorithms.* Baltimore, Maryland, 508–516. 60

BANSAL, N. AND HARCHOL-BALTER, M. 2001. Analysis of SRPT scheduling: Investigating unfairness. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '01).* 51, 55, 59, 64

BARFORD, P. AND CROVELLA, M. 1998. Generating representative web workloads for network and server performance evaluation. In *Proceedings* *of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '98).* Madison, WI, 151–160. 27

BATES, M. J. 1990. *The berry-picking search: User interface design.* Addison-Wesley. Edited by Harold Thimbleby. 19

BAUMOL, W. J. AND BLINDER, A. S. 1994. *Economics: Principles and Policy*, 6th ed. The Dryden Press — Harcourt, Brace & Company, Fort Worth, TX. 42, 46

BBC News 2004. When hi-tech meets high fantasy — BBC News. `http://news.bbc.co.uk/2/hi/technology/3672887.stm`. 21

BECKER, G. 1965. A theory of the allocation of time. *Economic Journal 75,* 299 (Sept.), 493–517. 15

BENDER, M. A., CHAKRABARTI, S., AND MUTHUKRISHNAN, S. 1998. Flow and stretch metrics for scheduling continuous job streams. In *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms (DA '98).* San Francisco, CA, 270–279. 51, 53

BENDER, M. A., MUTHUKRISHNAN, S., AND RAJARAMAN, R. 2002. Improved algorithms for stretch scheduling. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms.* 60

BENTHAM, J. 1823. *An introduction to the principles of morals and legislation*, corrected ed. W. Pickering, London. 118

Beowulf 2003. Beowulf.Org — The Beowulf Cluster Site. `http://www.beowulf.org/`. 43

BERMAN, F., FOX, G., AND HEY, T. 2003. *Grid Computing: Making the Global Infrastructure a Reality.* John Wiley & Sons. 37, 43

BESTAVROS, A. 1996. Speculative data dissemination and service. In *Proceedings of the International Conference on Data Engineering (ICDE '96).* New Orleans, LA. 27

Biowulf 2004. Using BLAST on biowulf. `http://biowulf.nih.gov/apps/blast/index.html`. 20, 71, 144

BLACK, D. 1991. Processors, priority, and policy: Mach scheduling for new environments. In *Proceedings of the USENIX 1991 Winter Conference.* 1–12. 107

BLACK, D. L. 1990. Scheduling support for concurrency and parallelism in the Mach operating system. *IEEE Computer 23,* 5 (May), 35–43. 37

BLAZE, M. 1992. NFS tracing by passive network monitoring. In *Proceedings of the 1992 USENIX Winter Conference.* San Francisco, CA. 115

BUBENIK, R. AND ZWAENEPOEL, W. 1989. Performance of optimistic make. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '89).* Berkeley, CA, 39–48. 27, 30, 104, 143

BUBENIK, R. AND ZWAENEPOEL, W. 1990. Semantics of optimistic computation. In *Proceedings of the 10th IEEE International Conference on Distributed Computing Systems (ICDCS '90).* Paris, France. 32

BURGER, J. AND GOCHFELD, M. 1998. The tragedy of the commons. *Environment 40,* 10 (Dec.), 4–13; 26–27. 120

CAO, P., FELTEN, E., AND LI, K. 1994. Application-controlled file caching policies. In *Proceedings of the Summer 1994 USENIX Conference.* Boston, MA, 171–182. 78

CHANG, F. AND GIBSON, G. A. 1999. Automatic I/O hint generation through speculative execution. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI '99).* New Orleans, LA, 1–14. 32

CHANG, F. W. 2001. Using speculative execution to automatically hide I/O latency. Ph.D. thesis, School of Computer Science, Carnegie Mellon University. Available as the technical report CMU-CS-01-172. 36

CHEN, C. M. AND ROUSSOPOULOS, N. 1994. Adaptive selectivity estimation using query feedback. In *Proceedings of the 1994 ACM International Conference on Management of Data (SIGMOD '94).* Minneapolis, MN, 161–172. 72

CHRISOCHOIDES, N., FEDOROV, A., LOWEKAMP, B. B., ZANGRILLI, M., AND LEE, C. 2003. A case study of optimistic computing on the grid: Parallel mesh generation. In *Proceedings of the Next Generation Systems Program Workshop of the International Parallel and Distributed Processing Symposium (NGS / IPDPS '03).* Nice, France. 32, 36

Condor 2003. Condor project homepage. `http://www.cs.wisc.edu/condor/`. 43, 44, 67, 103, 110, 223, 227, 228, 230, 231, 232

Condor DAGMan 2004. Condor DAGMan. `http://www.cs.wisc.edu/condor/dagman/`. 86, 230, 231

Conway, R. W., Maxwell, W. L., and Miller, L. W. 1967. *Theory of Scheduling.* Addison-Wesley Publishing Company, Reading, MA. 53, 59, 96

Corbató, F. J., Merwin-Daggett, M., and Daley, R. C. 1962. An experimental time-sharing system. In *Proceedings of the 1962 Spring Joint Computer Conference of the American Federation of Information Processing Societies (AFIPS).* 335–344. 67, 109

Cormen, T. H., Leiserson, C. E., and Rivest, R. L. 1990. *Introduction to Algorithms.* McGraw-Hill Book Company. 98, 101, 103, 229

Cowan, C. and Lutfiyya, H. 1995. Formal semantics for expressing optimism: The meaning of hope. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC '95).* Ottawa, Ontario, Canada, 164–173. 32

Crovella, M. E. 2000. Performance evaluation with heavy tailed distributions. *Lecture Notes in Computer Science 1786*, 1–9. 24, 141, 142

Crovella, M. E. and Bestavros, A. 1995. Explaining world wide web traffic self-similarity. Tech. Rep. bucs-tr-1995-015, Department of Computer Science, Boston University. Oct. 24, 27, 117

Crovella, M. E., Harchol-Balter, M., and Murta, C. D. 1997. Task assignment in a distributed system: Improving performance by unbalancing load. Tech. Rep. bucs-tr-1997-018, Department of Computer Science, Boston University. Oct. 142

Crovella, M. E. and Lipsky, L. 1997. Long-lasting transient conditions in simulations with heavy-tailed workloads. In *Proceedings of the 29th Winter Simulation Conference.* Atlanta, GA, 1005–1012. 142

Dawes, R. M. 1979. The robust beauty of improper linear models in decision making. *American Psychologist 34,* 7 (July), 571–582. Possibly influenced Yo La Tengo — Chris Stamey & Kirk Ross [1995]. 77, 110

DeGroot, D. 1990. Throttling and speculating on parallel architectures. In *Proceedings of Parbase '90.* Abstract of keynote speech. 15, 19, 22, 24, 77, 78, 104, 123

DEMERS, A., KESHAV, S., AND SHENKER, S. 1989. Analysis and simulation of a fair queueing algorithm. In *Proceedings of the Symposium on Communications, Architectures, and Protocols (SIGCOMM '89).* Austin, Texas, 1–12. 111

DESHPANDE, M. AND KARYPIS, G. 2000. Selective Markov models for predicting web-page accesses. Tech. Rep. 00-056, Department of Computer Science / Army HPC Research Center, University of Minnesota. Oct. 33

DiskSim 2004. The DiskSim simulation environment. `http://www.pdl.cmu.edu/DiskSim/`. 23

DOUGLIS, F. AND OUSTERHOUT, J. 1991. Transparent process migration: Design alternatives and the Sprite implementation. *Software—Practice & Experience 21,* 8 (Aug.), 757–785. 28, 67, 118

ECE 2002. Survey of twelve graduate students in the Electrical and Computer Engineering Department at Carnegie Mellon University. Personal communication. 22, 71

EDS 2004. Application selective outsourcing from EDS. `http://www.eds.com/services_offerings/so_appsvs_outsourcing.shtml`. 44

EGGERT, L. R. 2004. Background use of idle resource capacity. Ph.D. thesis, University of Southern California. 31, 104

ENDO, Y., WANG, Z., CHEN, J. B., AND SELTZER, M. 1996. Using latency to evaluate interactive system performance. In *Proceedings of the 2rd USENIX Symposium on Operating Systems Design and Implementation (OSDI '96).* Seattle, WA, 185–199. 53

EPPS, D. 2004. Personal communication. R&D director at Tippett Studio since 1992. 21, 29, 70, 144

ESSICK, R. B. 1990. An event-based fair share scheduler. In *Proceedings of the Winter 1990 USENIX Conference.* 147–162. 111

FAYYAD, U. 1998. Taming the giants and the monsters: Mining large databases for nuggets of knowledge. *Database Programming and Design 11,* 3 (Mar.). 24

FEITELSON, D. G. AND JETTE, M. A. 1997. Improved utilization and responsiveness with gang scheduling. In *Proceedings of the 3rd Workshop on Job Scheduling Strategies for Parallel Processing (IPPS / SPDP '97).*

Geneva, Switzerland, 238–261. Lecture Notes in Computer Science, vol. 1291, Springer-Verlag. 40, 53, 64, 65, 69, 141

Feitelson, D. G., Rudolph, L., Schwiegelshohn, U., Sevcik, K. C., and Wong, P. 1997. Theory and practice in parallel job scheduling. In *Proceedings of the 3rd Workshop on Job Scheduling Strategies for Parallel Processing (IPPS / SPDP '97)*. Geneva, Switzerland, 1–34. Lecture Notes in Computer Science, vol. 1291, Springer-Verlag. 27, 29, 69, 75, 89, 96

Fisher, D. 2002. Mozilla: Link prefetching FAQ. `http://mozilla.org/projects/netlib/Link_Prefetching_FAQ.html`. 27, 33

Foster, I. and Kesselman, C. 2004. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann. 37, 43

Foster, I., Kesselman, C., and Tuecke, S. 2001. The anatomy of the Grid: Enabling scalable virtual organizations. *International Journal of High Performance Computing Applications*. 43

FreeBSD 2004. The FreeBSD project. `http://www.freebsd.org/`. 103, 107, 110

Ganger, G. R. and Patt, Y. N. 1998. Using system-level models to evaluate I/O subsystem designs. *IEEE Transactions on Computers*, 667–678. 79

Gauss, C. F. 1821. Theoria combinationis observationum erroribus minumum obnoxiae. *Royal Society of Göttingen*. Reprinted by siam Classics in Applied Mathematics, 1995, with English translation by G.W. Stewart. 70

Genome. 2001. Initial sequencing and analysis of the human genome. *Nature 409*, 860–921. The Genome International Sequencing Consortium. 19

Ghormley, D. P., Petrou, D., Rodrigues, S. H., Vahdat, A. M., and Anderson, T. E. 1998. GLUnix: A global layer Unix for a network of workstations. *Software—Practice & Experience 28*, 9 (July), 929–961. 44, 66

Gibbons, R. 1997. A historical application profiler for use by parallel schedulers. In *Proceedings of the 3rd Workshop on Job Scheduling Strategies for Parallel Processing (IPPS / SPDP '97)*. Geneva, Switzerland. Lecture Notes in Computer Science, vol. 1291, Springer-Verlag. 214

GIBBS, W. W. 1997. Gordon E. Moore — Part 2. *Scientific American.com*. See `http://www.sciam.com/article.cfm?articleID=000C8D8B-7E63-1CDA-B4A8809EC588EEDF`. 15

GIBSON, G. AND CORBETT, P. 2004. pNFS problem statement. Internet-Draft Version 01, IETF. July. 38

GIBSON, G., NAGLE, D., AMIRI, K., BUTLER, J., CHANG, F., GOBIOFF, H., HARDIN, C., RIEDEL, E., ROCHBERG, D., AND ZELENKA, J. 1998. A cost-effective, high-bandwidth storage architecture. In *Proceedings of the 8th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '98)*. San Jose, CA. 38

GIDDINGS, M. AND KNUDSON, D. 2004. Xgrid-users email list. Subjects 'differing resources' and 'What's on your Xgrid' at `http://lists.apple.com/mailman/listinfo/xgrid-users`. 20, 144

Globus 2003. The Globus toolkit. `http://www-unix.globus.org/toolkit/`. 44, 66

GOLDING, R., BOSCH, P., STAELIN, C., SULLIVAN, T., AND WILKES, J. 1995. Idleness is not sloth. In *Proceedings of the Winter 1995 USENIX Conference*. New Orleans, 201–212. 104

GOODHEART, B. AND COX, J. 1994. *The Magic Garden Explained: The Internals of UNIX System V Release 4, an Open Systems Design*. Prentice-Hall. 107

GRAY, J. 2004. Distributed computing economics. `http://www.clustercomputing.org/content/tfcc-5-1-gray.html`. 38

HALSTEAD, JR., R. H. 1985. Multilisp: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems (TOPLAS '85) 7,* 4 (Oct.), 501–538. 31

HARCHOL-BALTER, M. 1999. The effect of heavy-tailed job size distributions on computer system design. In *Proceedings of the ASA-IMS Conference on Applications of Heavy-tailed Distributions in Economics, Engineering and Statistics*. Washington, DC. 141

HARCHOL-BALTER, M. 2002. Task assignment with unknown duration. *Journal of the ACM (JACM '02) 49,* 2 (Mar.), 260–288. 40, 66

Harchol-Balter, M. 2003a. Personal communication. Scheduling theory professor in the Computer Science Department at Carnegie Mellon University. 28, 95, 97

Harchol-Balter, M. 2003b. Class notes for 15-849b, Theory of performance modeling. `http://www.cs.cmu.edu/~harchol/Perfclass/class.html`. Several well-known queuing theory results are stated and proved in these notes; seminal cites are also included. 54, 56, 60, 61, 62, 63, 65, 142, 146

Harchol-Balter, M., Crovella, M. E., and Murta, C. D. 1997. To queue or not to queue: When queuing is better than timesharing in a distributed system. Tech. Rep. bucs-tr-1997-017, Department of Computer Science, Boston University. Oct. 60

Harchol-Balter, M. and Downey, A. B. 1997. Exploiting process lifetime distributions for dynamic load balancing. *ACM Transactions on Computer Systems (TOCS '97) 15,* 3 (Aug.), 253–285. 54, 60, 63, 68, 141

Harchol-Balter, M., Sigman, K., and Wierman, A. 2002. Asymptotic convergence of scheduling policies with respect to slowdown. In *IFIP WG 7.3 International Symposium on Computer Modeling, Measurement and Evaluation (Performance '02).* Rome, Italy. 53

Hardin, G. 1968. The tragedy of the commons. *Science 162*, 1243–48. 118, 119

Heidemann, J. S. and Popek, G. J. 1994. File-system development with stackable layers. *ACM Transactions on Computer Systems 12,* 1 (Feb.), 58–89. 115

Hellerstein, J. L. 1993. Achieving service rate objectives with decay usage scheduling. *IEEE Transactions on Software Engineering 19,* 8 (Aug.), 813–825. 108

Hennessy, J. L., Patterson, D. A., and Goldberg, D. 2002. *Computer Architecture: A Quantitative Approach*, 3rd ed. Morgan Kaufmann. 15, 26, 31

Hewlett-Packard 2004. Virtualization services — hp services. `http://www.hp.com/hps/spotlight/index_virtualization.html`. 44

Hildebrand, D. and Honeyman, P. 2004. nfsv4 and high performance file systems: Positioning to scale. Tech. Rep. citi-04-02, Center for Information Technology Integration, University of Michigan. Sept. 38

Hillner, J. 2003. The wall of fame. *Wired Magazine 11,* 12. 21, 43, 144

Hoffman, T. 2003. HP takes new pricing path for utility-based computing. Computerworld, `http://www.computerworld.com/managementtopics/management/itspending/story/0,10801,81522,00.html?nas=AM-81522`. 46

Holliman, D. 2003. Personal communication. Former system administrator for the Berkeley Phylogenomics Group. 20, 43

Huston, L., Sukthankar, R., Wickremesinghe, R., Satyanarayanan, M., Ganger, G., Riedel, E., and Ailamaki, A. 2003. Diamond: A storage architecture for early discard in interactive search. Tech. Rep. irp-tr-03-09, Intel Research Pittsburgh. Oct. 24

IBM 2004. ibm global services — Outsourcing / Hosting. `http://www-1.ibm.com/services/us/index.wss/it/so/a1000414`. 44

Jefferson, D. R. 1985. Virtual time. *ACM Transactions on Programming Languages and Systems (TOPLAS '85) 7,* 3 (July), 404–425. 32

Jensen, E. D., Locke, C. D., and Tokuda, H. 1985. A time-driven scheduling model for real-time operating systems. In *Proceedings of the 6th IEEE Real-Time Systems Symposium (RTSS '85)*. 112–122. 95, 112

Jones, M. B., Roşu, D., and Roşu, M.-C. 1997. cpu reservations and time constraints: Efficient, predictable scheduling of independent activities. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*. Saint Malo, France. 112

Kahneman, D. and Tversky, A. 1979. Prospect theory: An analysis of decision under risk. *Econometrica 47,* 2 (Mar.), 263–292. 80

Kapadia, N. H., Fortes, J. A. B., and Brodley, C. E. 1999. Predictive application-performance modeling in a computational grid environment. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing (HPDC '99)*. Redondo Beach, CA, 47–54. 72

KARP, R. M. AND RABIN, M. O. 1987. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development 31,* 2 (Mar.), 249–260. 19

KIM, M. AND NOBLE, B. 2001. Mobile network estimation. In *Proceedings of the ACM Conference on Mobile Computing and Networking.* Rome, Italy. 73

KING, J. 1993. The unfolding puzzle of protein folding. *Technology Review 96,* 4 (May/June), 54–61. 19

KOTZ, D. 1997. Disk-directed I/O for MIMD multiprocessors. *ACM Transactions on Computer Systems (TOCS '97) 15,* 1 (Feb.), 41–74. 78

LAMPSON, B. 1983. Hints for computer system design. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles (SOSP '83).* Bretton Woods, NH, 33–48. 77

LARMOUTH, J. 1978. Scheduling for immediate turnaround. *Software—Practice & Experience 8,* 5 (Sept.), 559–578. 111

LEE, C. A. 2002. Optimistic grid computing. Talk given at the *Performance Analysis and Distributed Computing Workshop (PADC '02),* Schloss Dagstuhl, Wadern, Germany. 32

Legion 2004. Legion: A worldwide virtual computer. `http://www.cs.virginia.edu/~legion/`. 44

Lemieux 2003. The Lemieux Supercomputer. `http://www.psc.edu/machines/tcs/lemieux.html`. 46, 63, 66, 81

LIVNY, M. 2004. Personal communication. Principal investigator of the Condor Project. 231

LOKOVIC, T. 2004. Personal communication. Graphics software engineer at Pixar Animation Studios since 1998. 21, 29, 144

LOPEZ, J. C. 2002. Personal communication. Graduate student in the Electrical and Computer Engineering Department at Carnegie Mellon University. 43, 64

MACKIE-MASON, J. K. AND VARIAN, H. R. 1995. Pricing congestible network resources. *IEEE Journal on Selected Areas in Communications (J-SAC '95) 13,* 7 (Sept.), 1141–1149. 46, 120

Maya Association 2004. The lord of the rings — Maya Association. `http://www.mayaassociation.fsbusiness.co.uk/mov-lor.htm`. 21, 22

McKusick, M. K., Bostic, K., Karels, M. J., and Quarterman, J. S. 1996. *The Design and Implementation of the 4.4BSD Operating System.* Addison-Wesley Publishing Company, Inc. 107, 108

Mercer, C. W. 1992. An introduction to real-time operating systems: Scheduling theory. `http://www.cs.cmu.edu/afs/cs/project/rtmach/public/papers/sur1.review.ps`. 112

Moore, G. E. 1965. Cramming more components onto integrated circuits. *Electronics 38,* 8 (Apr.). 15, 17

Musliner, D. J., Durfee, E. H., and Shin, K. G. 1992. Any-dimension algorithms. In *Proceedings of the 9th IEEE Workshop on Real-Time Operating Systems and Software (RTOSS '92).* Charlottesville, VA, 78–81. 16

Mutka, M. W. and Livny, M. 1987. Profiling workstations' available capacity for remote execution. In *Proceedings of the 12th IFIP WG 7.3 International Symposium on Computer Performance Modelling, Measurement, and Evaluation (Performance '87).* Brussels, Belgium, 529–544. 28

Narayanan, D. 2002. Operating system support for mobile interactive applications. Ph.D. thesis, School of Computer Science, Computer Science Department, Carnegie Mellon University. Available as the technical report cmu-cs-02-168. 42, 72

Narayanan, D., Flinn, J., and Satyanarayanan, M. 2000. Using history to improve mobile application adaptation. In *Proceedings of the 3rd IEEE Workshop on Mobile Computing Systems and Applications (WMCSA '00).* Monterey, CA. 70, 71, 72

NAS 2002. nas system documentation. `http://www.nas.nasa.gov/User/Systemsdocs/systemsdocs.html`. Contains links to the scheduling and usage policies of the computers at the nasa Advanced Supercomputer Center. 64, 65, 69

NEOS 2004. Neos server for optimization. `http://www-neos.mcs.anl.gov/neos/`. 230

Neugebauer, R. 1999. How elastic are real applications? In *Proceedings of the 9th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV '99)*. Basking Ridge, NJ, 197–200. Position paper. 124

Newell, A. 1990. *Unified Theories of Cognition*. Harvard University Press, Cambridge, MA. 124

Nieh, J. and Lam, M. S. 1997. The design, implementation and evaluation of smart: A scheduler for multimedia applications. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*. Saint Malo, France. 112

Nielsen, J. 1994. *Usability Engineering*. Morgan Kaufmann, San Francisco, CA. 123, 124

NS 2004. The network simulator — ns-2. `http://www.isi.edu/nsnam/ns/`. 23

O'Day, V. L. and Jeffries, R. 1993. Orienteering in an information landscape: How information seekers get from here to there. In *Proceedings of the 1993 Conference on Human Factors in Computing Systems (InterCHI)*. Amsterdam, Holland, The Netherlands, 438–445. 13

Osborne, R. B. 1990. Speculative computation in Multilisp. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*. Nice, France, 198–208. 31

Padmanabhan, V. N. and Mogul, J. C. 1996. Using predictive prefetching to improve world wide web latency. *ACM SIGCOMM Computer Communication Review (CCR '96) 26,* 3 (July), 22–36. 27, 33, 123

Parekh, S., Gandhi, N., Hellerstein, J., Tilbury, D., Jayram, T. S., and Bigus, J. 2001. Using control theory to achieve service level objectives in performance management. In *Proceedings of the Real-Time Systems Journal.* 126

Parsons, E. W. and Sevcik, K. C. 1997. Implementing multiprocessor scheduling disciplines. In *Proceedings of the 3rd Workshop on Job Scheduling Strategies for Parallel Processing (IPPS / SPDP '97)*. Geneva, Switzerland, 166–192. Lecture Notes in Computer Science, vol. 1291, Springer-Verlag. 230

PARSONS, I., UNRAU, R., SCHAEFFER, J., AND SZAFRON, D. 1997. PI/OT, Parallel I/O templates. *Parallel Computing 23,* 4–5 (June), 543–570. 78

PASHIGIAN, B. P., PELTZMAN, S., AND SUN, J.-M. 2003. Firm responses to income inequality and the cost of time. *Review of Industrial Organization 22,* 4 (June), 253–272. 15

PATTERSON, R. H. 2004. Personal communication. Author of the dissertation entitled *Informed prefetching and caching* [Patterson III, 1997]. 104

PATTERSON, R. H., GIBSON, G. A., GINTING, E., STODOLSKY, D., AND ZELENKA, J. 1995. Informed prefetching and caching. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95).* Copper Mountain Resort, CO, 79–95. 25, 26, 32, 78, 126, 128

PATTERSON III, R. H. 1997. Informed prefetching and caching. Ph.D. thesis, School of Computer Science, Carnegie Mellon University. Available as the technical report CMU-CS-97-204. 77, 78, 104, 258

PAXSON, V. AND FLOYD, S. 1994. Wide-area traffic: the failure of Poisson modeling. In *Proceedings of the Symposium on Communications, Architectures, and Protocols (SIGCOMM '89).* London, UK, 257–268. 141, 225

PEARSON, W. AND LIPMAN, D. 1988. Improved tools for biological sequence comparison. In *Proceedings of the National Academy of Sciences of the USA*. Vol. 85. 2444–2448. 20

PEREIRA, F. 2003. Personal communication. Graduate student in the Computer Science Department at Carnegie Mellon University. 43, 71

PETROU, D. 2002. Matching resource supply and resource demand. Thesis proposal, Department of Electrical and Computer Engineering, Carnegie Mellon University. 73

PETROU, D., GHORMLEY, D. P., AND ANDERSON, T. E. 1996. Predictive state restoration in desktop workstation clusters. Tech. Rep. CSD-96-921, Computer Science Department, University of California, Berkeley. Nov. 27, 118

PETROU, D., MILFORD, J. W., AND GIBSON, G. A. 1999. Implementing lottery scheduling: Matching the specializations in traditional schedulers. In *Proceedings of the USENIX 1999 Annual Technical Conference.* Monterey, California. 111

PFISTER, G. F. 1995. *In Search of Clusters.* Prentice Hall, Upper Saddle River, NJ. 42

Platform 2003. Platform Computing — Products — Platform LSF. `http://www.platform.com/products/LSF/`. 21, 44, 230

POLYZOTIS, N. AND IOANNIDIS, Y. 2003. Speculative query processing. In *Proceedings of the 1st Biennial Conference on Innovative Data Systems Research (CIDR '03).* Asilomar, CA. 31

PRABHAKAR, E. 2004. Personal communication. Employee of Apple Computer, Corp. 231

PVM 2004. PVM: Parallel virtual machine. `http://www.csm.ornl.gov/pvm/pvm_home.html`. 44

RAMAN, V. AND HELLERSTEIN, J. M. 2002. Partial results for online query processing. In *Proceedings of the 2002 ACM International Conference on Management of Data (SIGMOD '02).* Madison, WI, 275–286. 17

ROMER, P. 2000. Time: It really is money. *InformationWeek.* See `http://www.informationweek.com/803/romer.htm`. 15

RUEMMLER, C. AND WILKES, J. 1994. An introduction to disk drive modeling. *Computer 27,* 3 (Mar.), 17–28. 104

SARGENT, R. G. 1999. Validation and verification of simulation models. In *Proceedings of the 1999 Winter Simulation Conference.* Phoenix, AZ. 145, 222, 229

SCHLESINGER, S. AND OTHERS. 1979. Terminology for model credibility. *Simulation 32,* 3 (Jan.), 103–104. 145

SCHRAGE, L. E. 1968. A proof of the optimality of the shortest remaining processing time discipline. *Operations Research 16*, 678–690. 59

SCHROEDER, B. AND HARCHOL-BALTER, M. 2000. Evaluation of task assignment policies for supercomputing servers: The case for load unbalancing and fairness. In *Proceedings of the 9th IEEE Symposium on High Performance Distributed Computing (HPDC '00).* Pittsburgh, PA, 211–220. 42, 43

SCULLEY, J. 1989. The relationship between business and higher education: a perspective on the 21st century. *Communications of the ACM (CACM '89) 32,* 9 (Sept.), 1056–1061. 24

Shepler, S., Callaghan, B., Robinson, D., Thurlow, R., Beame, C., Eisler, M., and Noveck, D. 2000. nfs version 4 protocol. RFC 3010, IETF. December. 115

Shneiderman, B. 1997. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, 3rd ed. Addison-Wesley Publishing Company. 55

Shreedhar, M. and Varghese, G. 1996. Efficient fair queueing using deficit round-robin. *IEEE/ACM Transactions on Networking 4,* 3 (June), 375–385. 111

Simplescalar 2004. SimpleScalar llc. `http://www.simplescalar.com/`. 23, 43

Smith, A. 1776. *An inquiry into the nature and causes of the wealth of nations.* Whitestone, Dublin. 119

Smith, T. and Waterman, M. 1981. Identification of common molecular subsequences. *Journal of Molecular Biology 147,* 1 (Mar.), 195–197. 19

Smith, W. and Wong, P. 2002. Resource selection using execution and queue wait time predictions. Tech. Rep. nas-02-003, Computer Sciences Corporation, nasa Ames Research Center. July. 69

Spring, N. and Wolski, R. 1998. Application level scheduling of gene sequence comparison on metacomputers. In *Proceedings of the 12th ACM International Conference on Supercomputing (SC '98).* Melbourne, Australia, 141–148. 20, 70

Stankovic, J. A., Spuri, M., Natale, M. D., and Buttazzo, G. C. 1995. Implications of classical scheduling results for real-time systems. *IEEE Computer 28,* 6. 97, 113

Steere, D. C. 1997. Exploiting the non-determinism and asynchrony of set iterators to reduce aggregate file I/O latency. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97).* Saint Malo, France. 16, 24, 27, 32, 33, 78, 86

Steere, D. C., Goel, A., Gruenberg, J., McNamee, D., Pu, C., and Walpole, J. 1999. A feedback-driven proportion allocator for real-rate scheduling. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI '99).* New Orleans, LA, 145–158. 126

STRAATHOF, J. H., THAREJA, A. K., AND AGRAWALA, A. K. 1986. UNIX scheduling for large systems. In *Proceedings of the USENIX 1986 Winter Conference*. 111–139. 108

SUN, J., SHINJO, Y., AND ITANO, K. 1999. The implementation of a distributed file system supporting the parallel world model. In *Proceedings of the 3rd International Workshop on Advanced Parallel Processing Technologies*. Changsha, China, 43–47. 31, 36

Sun Grid 2003. Sun ONE Grid Engine Software. `http://wwws.sun.com/software/gridware/`. 44

TANENBAUM, A. S. 1992. *Modern Operating Systems*. Prentice Hall, New Jersey. 107, 118

TAUB, E. A. June 3, 2003. The 'Matrix' invented: A world of special effects. *The New York Times*, Late Edition — Final, Section C, Page 1. 21

TENNENHOUSE, D. 2000. Proactive computing. *Communications of the ACM 43,* 5 (May), 43–50. 17

The Open Group 1997. Systems management: Data storage manement (XDSM) API. `http://www.opengroup.org/onlinepubs/9695979099/toc.pdf`. 115

Think Secret 2004. Apple to bring cluster rendering to full line of video apps. `http://www.thinksecret.com/news/prorendering.html`. 21

THOMASSON, W. A. 2004. Unraveling the mystery of protein folding. `http://www.faseb.org/opar/protfold/protein.html`. 19

Top500 2004. Top500 supercomputer sites. `http://www.top500.org`. 37, 43

Toy Story 2004. Toy story. `http://www.pixar.com/featurefilms/ts/`. 21

TURING, A. M. 1936. On computable numbers, with an application to the entscheidungsproblem. *London Mathematical Society 2,* 42, 230–265. 49

UK Computing Research Committee 2004. Grand challenges for computing research — UK Computing Research Committee. `http://www.nesc.ac.uk/esi/events/Grand_Challenges/index.html`. 17

Valhalia, U. 1995. *UNIX Internals: The New Frontiers.* Prentice Hall. 107, 112

Venkataramani, A., Kokku, R., and Dahlin, M. 2002. tcp Nice: A mechanism for background transfers. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI '02).* Boston, MA. 33

Virginia Tech 2004. Terascale cluster — Research computing. `http://computing.vt.edu/research_computing/terascale/`. 43

Waldspurger, C. A. and Weihl, W. E. 1994. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation (OSDI '94).* 1–11. 111

Waldspurger, C. A. and Weihl, W. E. 1995. Stride scheduling: Deterministic proportional-share resource management. Tech. Rep. mit/lcs/tm-528, MIT Laboratory for Computer Science, Massachusetts Institute of Technology. June. 111

Waldspurger, C. A. and Weihl, W. E. 1996. An object-oriented framework for modular resource management. In *Proceedings of the 5th International Workshop on Object Orientation in Operating Systems (IWOOS '96).* Seattle, WA, 138–143. 111

Walker, I. 2002. Who can afford to 'stand and stare'. Unpublished manuscript from the University of Warwick. 15, 124

Want, R., Hopper, A., Falcão, V., and Gibbons, J. 1992. The active badge location system. *ACM Transactions on Information Systems (TOIS) 10,* 1 (Jan.), 91–102. 118

Weidenhammer 2004. Outsourcing / Hosting from Weidenhammer. `http://www.hammer.net/`. 44

Weisstein, E. W. 2004a. Exponential distribution. `http://mathworld.wolfram.com/ExponentialDistribution.html`. From MathWorld — A Wolfram Web Resource. 140

Weisstein, E. W. 2004b. Hazard function. `http://mathworld.wolfram.com/HazardFunction.html`. From MathWorld — A Wolfram Web Resource. 59

WEISSTEIN, E. W. 2004c. Moment. `http://mathworld.wolfram.com/Moment.html`. From MathWorld — A Wolfram Web Resource. 142

WEISSTEIN, E. W. 2004d. Pareto distribution. `http://mathworld.wolfram.com/ParetoDistribution.html`. From MathWorld — A Wolfram Web Resource. 141

WEISSTEIN, E. W. 2004e. Poisson process. `http://mathworld.wolfram.com/PoissonProcess.html`. From MathWorld — A Wolfram Web Resource. 59

WEISSTEIN, E. W. 2004f. Uniform distribution. `http://mathworld.wolfram.com/UniformDistribution.html`. From MathWorld — A Wolfram Web Resource. 139

WEISSTEIN, E. W. 2004g. Variance. `http://mathworld.wolfram.com/Variance.html`. From MathWorld — A Wolfram Web Resource. 50

WEISSTEIN, E. W. 2004h. Variation coefficient. `http://mathworld.wolfram.com/VariationCoefficient.html`. From MathWorld — A Wolfram Web Resource. 60

WEISSTEIN, E. W. 2004i. Zipf's law. `http://mathworld.wolfram.com/ZipfsLaw.html`. From MathWorld — A Wolfram Web Resource. 24

WELSH, M., CULLER, D., AND BREWER, E. 2001. SEDA: An architecture for well-conditioned, scalable Internet services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01).* Chateau Lake Louise, Banff, Canada. 128

WENISCH, T. 2003. Personal communication. Graduate student in the Electical and Computer Engineering Department at Carnegie Mellon University. 18, 43, 71

WIERMAN, A. 2004. Personal communication. Scheduling theory graduate student researcher in the Computer Science Department at Carnegie Mellon University. 60, 61

WIERMAN, A., BANSAL, N., AND HARCHOL-BALTER, M. 2004. A note on comparing response times in M/GI/1/FB and M/GI/1/PS queues. *Operations Research Letters 32,* 1 (Jan.), 73–76. 60

Wikipedia 2004. Lazy evaluation. `http://en.wikipedia.org/wiki/Lazy_evaluation`. 30

WILLMOTT, A. J. 2000. Hierarchical radiosity with multiresolution meshes. Ph.D. thesis, School of Computer Science, Carnegie Mellon University. Available as the technical report CMU-CS-00-166. 71

WYCKOFF, P., JOHNSON, T., AND JEONG, K. 1998. Finding idle periods on networks of workstations. Tech. Rep. 761, NYU Computer Science Department. Mar. 118

Xcode 2004. Xcode. http://developer.apple.com/tools/xcode/. 30

Xgrid 2004. Apple — ACG — Xgrid. http://www.apple.com/acg/xgrid/. 23, 44, 115, 231

YO LA TENGO — CHRIS STAMEY & KIRK ROSS. 1995. The robust beauty of improper linear models in decision making. Compact disc, CD ESD 1995. See http://www.sunsquashed.com/net/Discog/robust.html. 249

ZIPF, G. 1932. *Selective studies and the principle of relative frequency in language.* Harvard University Press, Cambridge, MA. 24